NPS ARCHIVE
2000.03
HAZIR, Y.

KNOX LIBRARY
POSTGRADUATE SCHOOL
EREY CA 93943-5101

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION AND INTEGRATION OF THE OBJECT TRANSACTION SERVICE OF CORBA TO A JAVA APPLICATION DATABASE PROGRAM

by

Yildiray Hazir

March 2000

Thesis Advisor : C. Thomas Wu
Second Reader: Chris Eagle

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND IMPLEMENTATION AND INTEGRATION OF THE OBJECT TRANSACTION SERVICE OF CORBA TO A JAVA APPLICATION DATABASE PROGRAM.	5. FUNDING NUMBERS
6. AUTHOR Hazir, Yildiray	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT

In examining the recent trend of the Client / Server computing technology, it can be seen that distributed object technology is ready to take off. The CORBA (Common Object Request Broker) architecture is the most widely known and readily available candidate for development. For this reason CORBA has attracted our attention.The OMG(Object Management Group), a consortium of object venders, developed the CORBA standard in the fall of 1990 as a common interconnection bus for distributed objects. Transaction processing is useful not only in database applications but also in building robust mission-critical applications. Utilizing CORBA one can build reliable distributed software systems in a much easier way. CORBA is the most widely accepted standard in this field and there are many CORBA implementations available now. Moreover, the transaction concept is the key to ensure the reliability and availability of Client / Server applications.In this thesis transaction properties were applied to a database application program based on Naval Post Graduate School's Course Iteration System. For this purpose an Object Transaction Service was provided based upon the CORBA architecture. It takes advantage of object-oriented programming to help programmers implement transactional applications in a much easier way.In late 1994, the OMG also published the specification for the object transaction service. This specification is adopted as the blue print for this study. This thesis presents the implementation and integration of the object transaction service based on CORBA. JDBC (Java Database Connection) was not used for transaction property, because JDBC is currently limited in that it cannot manage transactions across multiple connections. For transaction support across databases or object services, CORBA's Transaction Service provides the best level of abstraction.

14. SUBJECT TERMS Software, Database, Distributed Object, Corba, OTS (Object Transaction Service),JDBC (Java Database Connectivity) and Java.	15. NUMBER OF PAGES 125
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

IMPLEMENTATION AND INTEGRATION OF THE OBJECT TRANSACTION SERVICE OF CORBA TO A JAVA APPLICATION DATABASE PROGRAM

Yildiray Hazir
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

ABSTRACT

In examining the recent trend of the Client / Server computing technology, it can be seen that distributed object technology is ready to take off. The CORBA (Common Object Request Broker) architecture is the most widely known and readily available candidate for development. For this reason CORBA has attracted our attention.

The OMG(Object Management Group), a consortium of object venders, developed the CORBA standard in the fall of 1990 as a common interconnection bus for distributed objects.

Transaction processing is useful not only in database applications but also in building robust mission-critical applications. Utilizing CORBA one can build reliable distributed software systems in a much easier way.

CORBA is the most widely accepted standard in this field and there are many CORBA implementations available now. Moreover, the transaction concept is the key to ensure the reliability and availability of Client / Server applications.

In this thesis transaction properties were applied to a database application program based on Naval Post Graduate School's Course Iteration System. For this purpose an Object Transaction Service was provided based upon the CORBA architecture. It takes advantage of object-oriented programming to help programmers implement transactional applications in a much easier way.

In late 1994, the OMG also published the specification for the object transaction service. This specification is adopted as the blue print for this study. This thesis presents the implementation and integration of the object transaction service based on CORBA.

JDBC (Java Database Connection) was not used for transaction property, because JDBC is currently limited in that it cannot manage transactions across multiple connections. For transaction support across databases or object services, CORBA's Transaction Service provides the best level of abstraction.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. OVERVIEW 1

B. BACKGROUND 3

C. PROBLEM STATEMENT 4

D. OBJECTIVE 5

E. SCOPE AND LIMITATIONS 6

F. ORGANIZATION OF THESIS 7

II. THE CORBA ENVIRONMENT 9

A. OMG's OBJECT REFERENCE MODEL 9

1. Organization of The Reference Model 9

2. Components of The Reference Model 10

3. Structure of an Object Request Broker 11

B. APPLICATION DEVELOPMENT IN CORBA 12

1. Concept of CORBA 13

2. Interface Definition Language 15

3. ORB Client 16

4. Object Implementation 17

5. ORB and RPC 18

6. Programming Environment 18

III. SYSTEM OVERVIEW 19

A. TRANSACTION DEFINITION 19

B. TRANSACTION PROPERTIES 19

1. Atomicity Property 19

2. Consistency Property 20

3. Isolation Property 21

4. Durability Property 21

C. TRANSACTIONAL CLIENT 22
D. TRANSACTIONAL SERVER 22
E. RECOVERABLE SERVER 24
F. OTS MANAGER 25
G. TRANSACTIONS MODELS 27
H. PROGRAMMING MODELS 27
1. Implicit Method 27
2. Explicit Method 28

IV. SYSTEM ARCHITECTURE 29

A. OVERVIEW 29
B. OBJECT TRANSACTION SERVICE INTERFACES 31
1. Current Interface 31
2. Coordinator Interface 32
3. Resource Interface 33
C. CLIENT AND SERVERS 33
1. Begin a Transaction 35
2. Transaction Coordinator 37
3. Transaction Participants 39
4. Processing of Two – Phase Commit Protocol 41

V. IMPLEMENTATION ISSUES 43

A. BACKGROUND REVIEW 43
B. OVERVIEW OF THE EXAMPLE 44
C. A JAVA CORBA SAMPLE 46
D. WRITING THE PROJECT "IDL" 47
E. INTERFACE DEFINITION LANGUAGE (IDL) TO JAVA MAPPING 48
1. Modules 48
2. Interfaces 49
3. Exceptions 49

4. Structures 49
5. Sequences 49
6. Strings 50
7. CORBA Parameters 50

F. WRITING THE TRANSACTION ORIGINATOR (CLIENT PROGRAM) . 50
1. Initializing the ORB 51
2. CosTransactions Service Initialization 51
3. Get the Transactional Server Reference from an Object File 51
4. Narrow the Object Reference 52
5. Get the Current Object and Begin a Transaction 53
6. Invoking Methods on the RegisterImpl class
from Client Program 54
7. Committing or Rolling Back the Transaction 56

H. WRITING THE SERVER PROGRAM 57
1. Initializing the ORB, the CosTransactions and
the BOA in the Server Program 57
2. Declare the Transactional Object and
Connect it to Basic Object adaptor 58
3. Convert the Transactional Object 's Reference to String and
Convert its Reference to String 58
4. Then Wait for Incoming Requests from the Clients 59

I. WRITING THE TRANSACTIONAL OBJECT (REGISTER) 60
1. Understanding the RegistrationImpl Class Hierarchy 60
2. Implementing the RegistrationImpl Object and its Methods 60
3. Implementing Methods of the RegistrationImpl Object 61
4. Implementing the Lock Object 64

VI. CONCLUSIONS 67

A. CORBA OBJECT TRANSACTION SERVICE 67

B. JAVA DATABASE CONNECTIVITY TRANSACTION SUPPORT 68

APPENDIX A: IDL DEFINITION FOR COSTRANSACTION 71

APPENDIX B: IDL DEFINITION FOR PROJECT 75

APPENDIX C: SERVER IMPLEMENTATION 77

APPENDIX D: CLIENT IMPLEMENTATION 79

APPENDIX E: REGISTRATION CLASS IMPLEMENTATION 89

APPENDIX F: LOCK CLASS IMPLEMENTATION 101

APPENDIX G: APPLICATION PROGRAM GUI SCREEN SHOTS 103

APPENDIX H: CLIENT \ SERVER DOS OUTPUT SCREEN SHOTS 105

APPENDIX I: DATABASE TABLES SCREEN SHOTS 107

LIST OF REFERENCES 109

INITIAL DISTRIBUTION LIST 111

LIST OF FIGURES

1. Figure 1.1 OMA Framework 5
2. Figure 1.2 System Sketch 6
3. Figure 2.1 The ORB Interconnection Bus 11
4. Figure 2.2 The Structure of a CORBA Object Request Broker 12
5. Figure 2.3 Common Object Request Broker Architecture 13
6. Figure 2.4 The Application Development in CORBA 14
7. Figure 2.5 The Operation Call on a Proxy 17
8. Figure 3.1 Object Transaction Service Overview 23
9. Figure 3.2 The Functional Diagram of the Applications and the OTS Manager 25
10. Figure 4.1 System Architecture of OTS 30
11. Figure 4.2.a The Register IDL Interface 34

    Figure 4.2.b The Programming Environment for Clients 35
12. Figure 4.3 Beginning a Transaction 36
13. Figure 4.4 The Propagation of Transaction Context 38
14. Figure 4.5 Communication in Two-Phase Commit Protocol 40
15. Figure 5.1 The Interfaces and Transaction Manager 44
16. Figure 5.2 Project IDL 47

I. INTRODUCTION

A. OVERVIEW

A thesis usually begins with a general review; instead I will ask some questions to make the subject more understandable after giving the general definition of thesis.

This thesis was based on a real application that is used by the Naval Post Graduate School course registration system. This implementation simulates that students from different curricular can enter their course requests to a database table instead of hard copy signup sheet. This system supports students concurrent attempts to update their account with a transaction property. Students' interaction was simulated to add a course from Course Table to Registered Course table and drop a course from the Registered Course table corresponding to the name of the students. This thesis implements and integrates the transaction service to this Course Registration Database application program. The following questions were asked to identify the definition of Transaction Service.

- What happens if a failure occurs during modification of resources?
- Which operations have been completed?
- Which operations have not (and have to be done again)?
- In which states will the resources be?

Whatever measures are taken to avoid failures, failures still occur. Power supplies may fail, disks crash and operating systems may not supply the stability we hope for. Assume that such a failure occurs while one component updates resources from belonging to other components. For example during course registration , while adding or dropping courses from the Course Registration Database. This database consists of two tables; a Courses table that lists available Computer Science courses and a Registered Course table that is updated when Computer Science students add and drop new courses at the beginning of each quarter. This action performs two primitive operation invocations: adding a new course from the Course table and dropping a course from the Registered Course table.

If a failure occurs during the operation, a number of questions arise:
Which actions have been completed and which have not? In the case of course updating, has the add operation been completed but the drop operation not completed? What is the state of the resources? Are there any operations necessary to restore their integrity? In the case of adding a course has the course that was on its way from one table to the other table been lost and does it need to be recovered?

The approach taken in both databases and distributed systems to deal with these problems are *transactions*. A *transaction* is a sequence operation that is either performed completely or not at all. If it is completed, the effect of a transaction is persistent and cannot be affected by failures.

"*Transactions* are more than just business events; they have become an applications design philosophy that guarantees robustness in distributed systems."[1] Traditionally, we use the transaction concept when we need to perform a set of atomic operations on a database to ensure the consistency of shared data. But the transaction concept is not only useful in the database domain. It is impossible to create a mission-critical client / server application without the need to update some data. Transactions have become common and desirable services to construct reliable and available client / server applications.

"The concept of transactions has been successfully applied in many commercial data management systems to build robust and reliable systems." [2] As network technology grows, the environment of distributed computing is becoming more and more mature. This motivates the introduction of transaction concepts into distributed environments. As we can see, this concept has dominated the world of database processing.

Taking a look at the recent trends in client / server computing technology, it is easy to see that distributed object technology is ready to take off. What is a distributed object?

If you have experience writing programs in an object-oriented language like C++ and Java, you will not be unfamiliar with the word "object" in an object-oriented world. An object is a functional unit of a program, which encapsulates code and data, and can be specialized by means of inheritance. Naturally, an object cannot reach across the compiled-language and address spaces in a traditional programming environment. It is clear that in order to introduce the object-oriented concept into client / server computing, we have to break these limitations.

A distributed object is a binary component that can be accessed from clients in remote hosts by means of method invocations through a common software bus. Clients have no idea what language was used to create the objects or even the physical locations of these server objects. Clients need only know the names and interfaces that server objects export to the software bus. So we can draw the picture: every component ties to the common software bus to publish their interfaces after which these components can interoperate with each other, forming the basis of client/server computing. "Distributed object technology inherits the advantage of object-oriented programming allowing large applications to be broken into small and manageable components that coexist on the intergalactic bus"[7]. We can say that distributed object technology promises the most flexible client / server computing model.

B. BACKGROUND

In the transaction processing world, all products support transaction services over heterogeneous platforms, however, none of them support the object-oriented paradigm. There are several different approaches to distributed object technology. Two of the most well known distributed object infrastructure standards are Microsoft's DCOM[3] and OMG's CORBA[4].

Each of them takes a different approach toward object distribution.[3,5] Compared to the CORBA object model, DCOM must be considered weaker since it:

- does not support inheritance of components
- is not as strongly typed
- does not support exceptions

"The OMG has adopted an internetworking specification between the two object models. It will be refined into an interoperability specification and enable DCOM objects to invoke services from CORBA objects and vice versa"[6]. The main difference is DCOM does not support the inheritance feature of object-oriented technology and originally aims at document processing. CORBA, on the other hand, is aimed at general purpose distributed computing.

A standard is necessary to pave the way toward success. The OMG (Object Management Group) specifies an architecture for an open software bus, called CORBA (Common Object Request Broker Architecture)[1]. CORBA which is proposed by OMG (Object Management Group) is the realization of OMA (Object Management Architecture), also proposed by OMG for distributed object computing. It defines a set of common services that are used to allow objects to invoke each other's methods without regard for the platform on which they operate, or the language in which they are written. Just as Java provides independence at the platform level CORBA provides independence at the language and platform level, as shown in Figure 1.1.

CORBA consists of four major parts: object request broker (ORB), common object service specification (COSS), common facility architecture (CFA) and application objects[1]. The ORB serves as a software bus that forwards messages between client and server objects. COSS defines a set of basic building blocks over the ORB. These building blocks are realized as server objects. COSS defines a number of services that are useful to any application in general. I chose to use CORBA as the distributed object infrastructure because it is an open standard and widely support in heterogeneous environments. Additionally the TNRL (Turkish Naval Research Lab) directed the use of CORBA for future projects.

C. PROBLEM STATEMENT

The OMG does not provide implementations for these services but provides the interfaces by which the services are offered. CFA specifies a few facilities that are closer to application level and are closer to a specific application domain.

CORBA defines the interfaces and functionalities of the ORB via which client objects may reuse COSS and CFA objects or access other server objects. CORBA specifies an ORB(Object Request Broker) through which a client can invoke the methods of remote objects either statically or dynamically. The OMG also defines an IDL (Interface Definition Language) to specify the interfaces between components and the software bus. IDL is independent of any programming language. From the client's point of view, what they see is an IDL interface that a server object exports and they need not know how the server object is implemented.

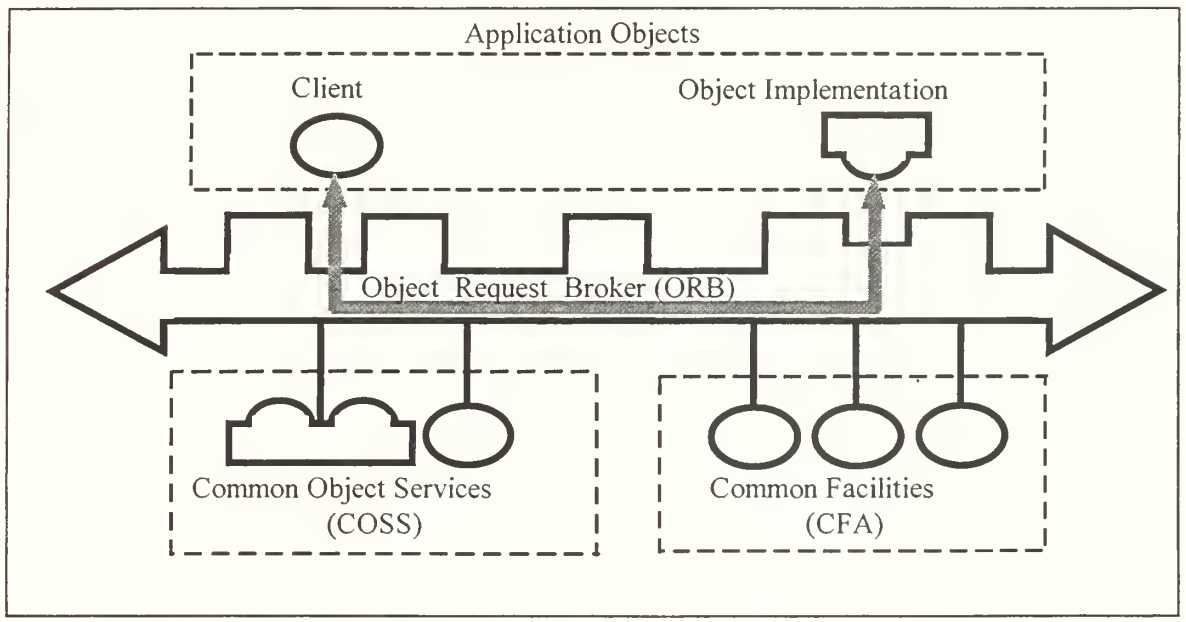


Figure 1.1 OMA Framework [From Ref. 6]

D. OBJECTIVE

As mentioned above, we believe distributed object technology is the future of client / server computing. It will make sense if we can provide some common services on top of the standard CORBA services to help with the development of distributed software. Because transactions are essential for building reliable distributed applications, an object transaction service was chosen as the first step toward distributed object computing. The main idea is to create an ordinary object and make it transactional by inheriting the interfaces defined by the transaction service. This enables it to participate in atomic transactions, even in face of failures.

The goal of this research is to explore and implement the Object Transaction Service (OTS) which is defined in COSS, on a Windows NT platform so that application object programmers can reuse these services to build reliable and robust distributed software efficiently. OTS supports not only the flat transaction model but also the nested transaction model. Additionally it defines a set of interfaces for recoverable objects, a kind of server object that can recover its states when suffering failures.

E. SCOPE AND LIMITATIONS

An application can take advantage of the OTS as follows (Figure 1.2). In the typical scenario, a client first begins a transaction by issuing a request to an object in OTS, which constructs a corresponding transaction context. The client then performs transactional operations by issuing requests to resource objects. The resource objects in turn register themselves with the OTS and acquire locks, in order to control concurrent access to shared data items.

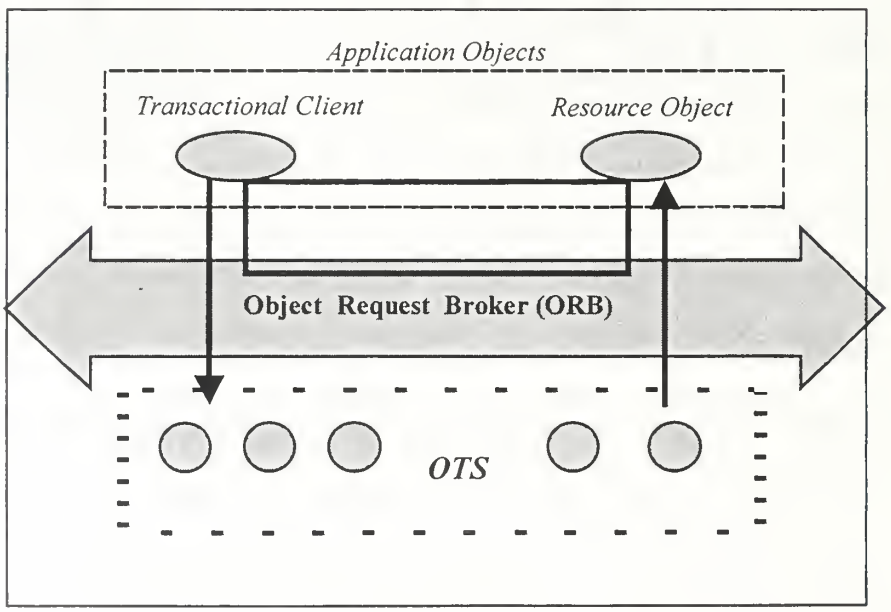


Figure 1.2 System Sketch [From Ref. 6]

Eventually, the client ends the transaction by issuing another request to the OTS which triggers the atomic commit procedure to synchronize all objects involved in this transaction on behalf of the client. Thus, OTS guarantees the essential ACID properties (detailed in the next chapter) with respect to the transaction.

The design of the service consists of three parts: the OTS manager and a suite of libraries that help the resource object programmer in creating a recoverable transactional object. Programmers who want to use transaction services should inherit the OTS interfaces (detailed in Chapter II). Programmers who want to provide objects to be used within transactions can use the libraries suite developed in this thesis. The current version of this libraries does support both flat and nested transaction models on the Windows NT platform.

F. ORGANIZATION OF THESIS

This thesis is organized as follows: Chapter II introduces the CORBA environment. Chapter III provides an overview of the system. The IDL interfaces of the Object Transaction Service and the system architecture is described in Chapter IV. Chapter V discusses the issues of implementing the Object Transaction Service. Finally a conclusion is presented in Chapter VI.

THIS PAGE INTENTIONALLY LEFT BLANK

II. THE CORBA ENVIRONMENT

Our Object Transaction Service is built upon the OMG's CORBA (Common Object Request Broker Architecture). In order to have a common base for further discussion, we review the essential background about OMG's CORBA and some related works. The development environment is also addressed.

A. OMG's OBJECT REFERENCE MODEL

The Object Management Group, Inc. (OMG) is an international organization concerned with object technologies. Its goal is to establish industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. OMG establishes the Object Management Architecture (OMA) to provide the conceptual infrastructure upon which all OMG specifications are based. It is known as OMA object Reference Model.

1. Organization of The Reference Model

The Reference Model identifies and characterizes the components, interfaces and protocols that compose OMG's Object Management Architecture, but does not itself define them in detail. Generally speaking, the Reference Model: [1]

- Identifies the major separable components of the total Object Management Architecture
- Characterizes the functions provided by each component
- Explains the relationships between the components and with the external operating environment
- Identifies the protocols and interfaces for accessing the components

Specifically, the Reference Model addresses:

- How objects make and receive requests and responses
- The basic operations that must be provided for every object
- Object interfaces that provide common facilities useful in many applications

2. Components of the Reference Model

The Reference Model consists of the following components: (Figure 1-1)

- Object Request Broker, which enables objects to transparently make and receive requests and responses in a distributed environment. In so doing, an ORB provides interoperability between applications on different machines in heterogeneous distributed environments and interconnects multiple object systems. Some details of how to use an ORB to develop distributed systems will be shown in the next section. [1]

- Object Services, a collection of services that support basic functions for using and implementing objects. These services are independent of application domains and act as the building blocks of distributed applications. The operations provided by Object Services are made available through the ORB. OTS is an example of Object Services. [6]

- Common Facilities, OMG defines a set of services that many applications may share, but which are not as fundamental as the Object Services. They are usually domain specific services. For instance, a printing and spooling system or electronic mail facility could be classified as a common facility.

- Application Objects, corresponds to the traditional notion of an application and are not standardized by OMG. It is important to realize that classes that fall into the Application Objects classification are at the same OMA semantic level as Common Facilities classes. For example, one can take advantage of Object Services and/or Common Facilities to build his own Application Object that can be shared through the network.

In general terms, the Application Objects and Common Facilities have an application orientation while the Object Request Broker and Object Services are concerned more with the system or infrastructure aspects of distributed object management. The Object Request Broker then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone can not enable interoperability at the application semantic level. It acts as a telephone exchange, which provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Object Services, Common Facilities and Application Objects provide different levels of semantics by using an ORB as the basis for communication.

3. Structure of an Object Request Broker

As described by OMG: "The Object Request Broker provides the mechanisms by which objects transparently make requests and receive responses.

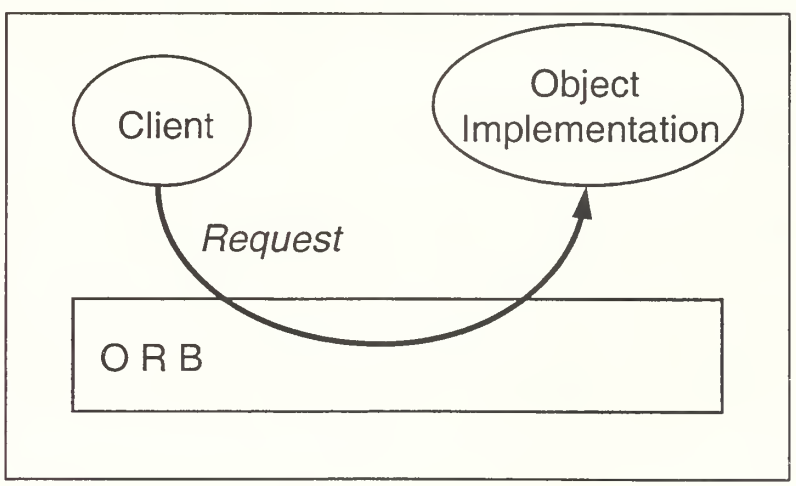


Figure 2.1 The ORB Interconnection Bus

The ORB provides interoperability between applications on different machines in heterogeneous distributed environment and seamlessly interconnects multiple object systems"[3]. Thus, we can view the ORB as an object interconnection bus through which objects in different machines can communicate with each other to complete the cooperative work. This is shown in Figure 2.1.

The ORB is responsible for finding, communicating with, and activating the object server. The structure of a CORBA Object Request Broker is given in Figure 2.2, and it is introduced in the succeeding subsections:

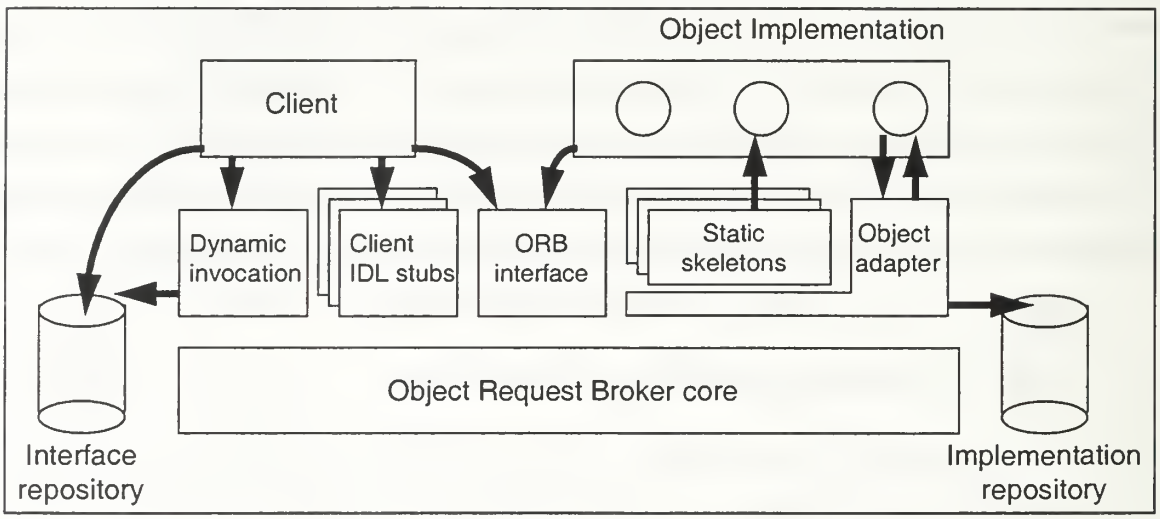


Figure 2.2 The Structure of a CORBA Object Request Broker [From Ref. 2]

B. APPLICATION DEVELOPMENT IN CORBA

As shown in Figure 2.3, CORBA environment consists of the following major components: the client object, the object server, the IDL compiler and the ORB. We briefly describe the functionality of each of these components and then discuss the interaction among these components during the development and the execution of the program.

In CORBA, the services of an object are specified in terms of an application program interface (API) using the standard CORBA interface definition language (IDL) which is machine and programming language independent. An object that implements the service (or API) according to an IDL interface specification is called an object implementation.

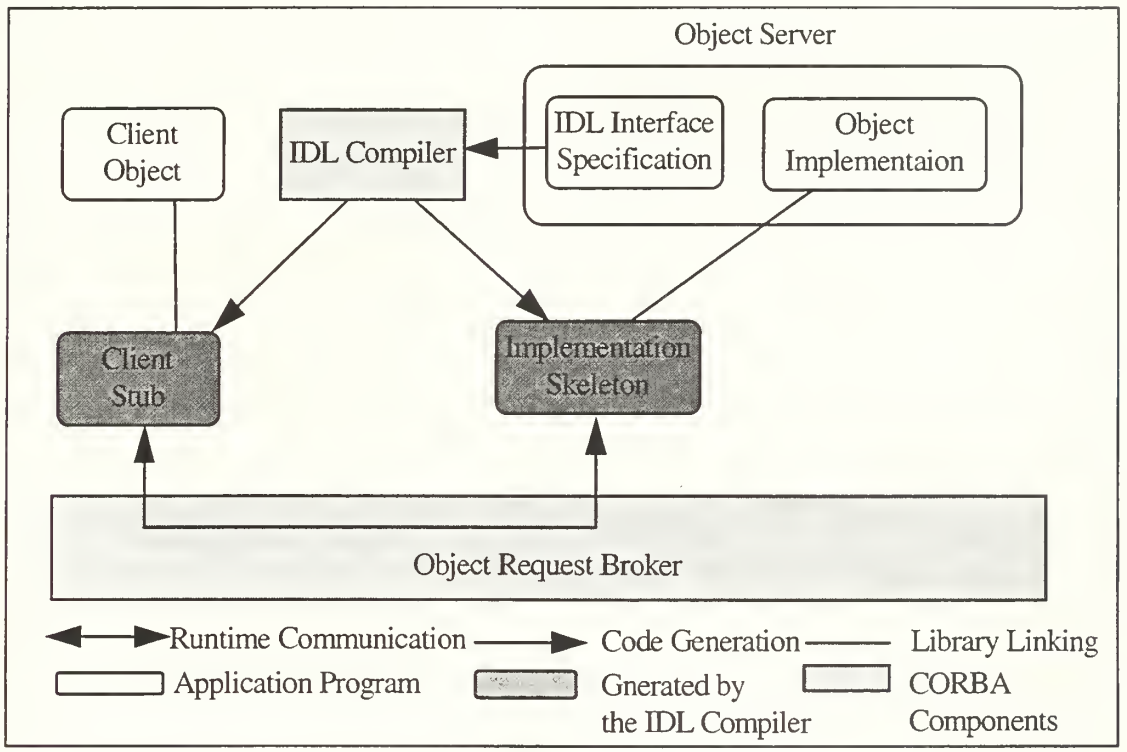


Figure 2.3 Common Object Request Broker Architecture

1. The Concept of CORBA

In another point of view, an object implementation is an executable entity that is capable of providing to the client, a service declared by the IDL. A client makes a request to an object implementation and expects the reply via an ORB. Here, the ORB serves as a software interconnection bus between the client and the object implementation. As shown in Figure 2.3, a client is able to access the services of an object implementation only if it has a reference to that object. This reference, called the client stub, is generated by the IDL compiler from the IDL specification of the object server.

The object implementation can provide its service over the networks via an ORB only if it has an implementation skeleton, which is also generated by the IDL compiler. Figure 2.4 illustrates the complete development of an object implementation and a client program.

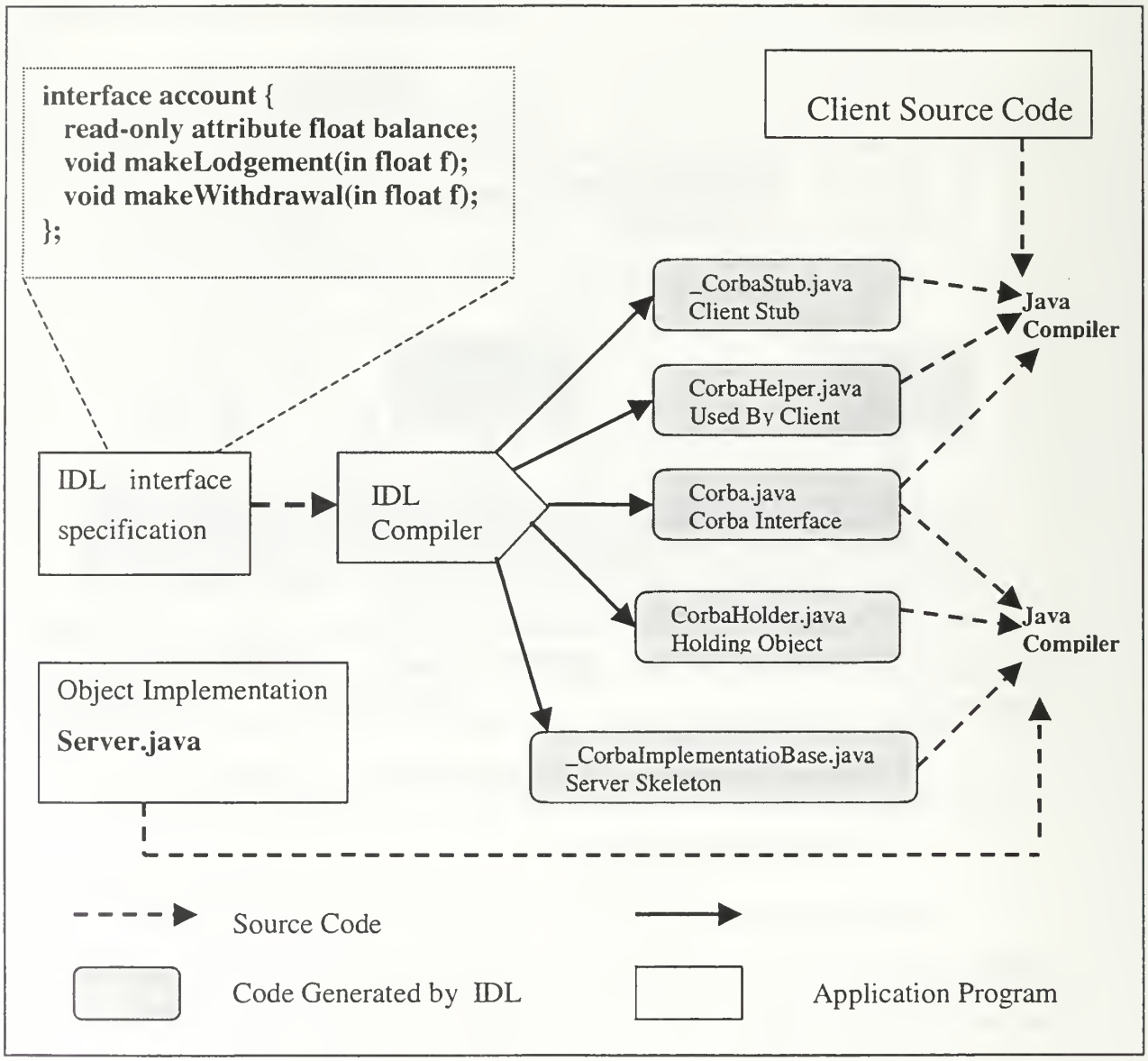


Figure 2.4 The Application Development in CORBA

Corba provides a mapping between a name and an object reference. Storing such a mapping in the ORB is known as binding an object, and removing this entry is called unbinding.

Obtaining an object reference which is bound to a name is known as resolving the name. We now briefly describe the basic concept of how CORBA and its components operate. To invoke an operation on a remote object implementation, a client must first bind to that object. The ORB then checks if an instance of the remote object implementation exists, if not, an instance of the object implementation will be created. The client can then invoke operations on the remote object implementation by sending requests.

The object implementation returns the results of the invocation to the client via the ORB after it completes the invocation. Subsequent invocations can be made without re-establishment of the communication channel.

Note that IDL is a definition language, it is not an operational programming language such as Java. Given an IDL interface, there may exist many object implementations for that interface. Moreover, these object implementations may be implemented in various programming languages on different operating systems or hardware platforms. IDL compilers are used to compile IDL interfaces, producing a client stub and and implementation skeleton in a specific target programming language according to CORBA language binding specifications. IDL compilers exist to map IDL to C, C++, Java and many other languages.

2. Interface Definition Language

Interface Definition Language (IDL) is the key to the success of CORBA, so we will take a look at it first. CORBA 1.1, introduced in 1991, defined the IDL. IDL provides a language to define the public interface for an object that will be accessible across the ORB, similar to defining an object in an object-oriented language. The interface of an object consists of named attributes, operations and the parameters required by these operations. Using IDL, an object publishes its interface to the common interconnection bus (ORB), allowing clients connected to the bus know what operations are provided and how these operations can be invoked across the bus. IDL is the contract that binds clients to server components.

Most importantly, an IDL interface is independent of the programming language that is used to implement it. From the client program's point of view, only the IDL interface of a server object is important, the choice of implementation language is not. As a result, the client can use the same semantics to access different objects implemented by different languages, through the ORB. As in most object-oriented languages, we also can use inheritance to define new IDL interfaces.

The following is an IDL interface example of a course registration object.

```
interface Register
{
    courseSeq add_course(in string course_index_number, in string Student_ID,
            in string password , in string studentName)
    courseSeq drop_course(in string course_index_number, in string Student_ID,
            in string password , in string studentName)
};
```

The above interface provides two operations: *add_course* and *drop_course* . Both of these operations have input parameters *course_index_number, Student_ID, password and studentName* .

3. ORB Client

Now, let us see how CORBA works on the client side. As Figure 2.2 shows, CORBA provides two ways for clients to invoke services. The first way is the static invocation in which client IDL stubs(generated by the IDL compiler) act as proxies to object implementations. In the context of the client program, the syntax of invoking a remote object is the same as invoking a local object. When invoking a remote object, the client requests are forwarded by the client side stubs to the remote object through the ORB.(Figure 2.5) Sometimes we don't know the services we want to invoke until run time. Thus, it is not possible to include the IDL stub code in our client code. In such a situation, dynamic invocation APIs can help us discover a service that we want to invoke and its definition, then we can issue a request to it.

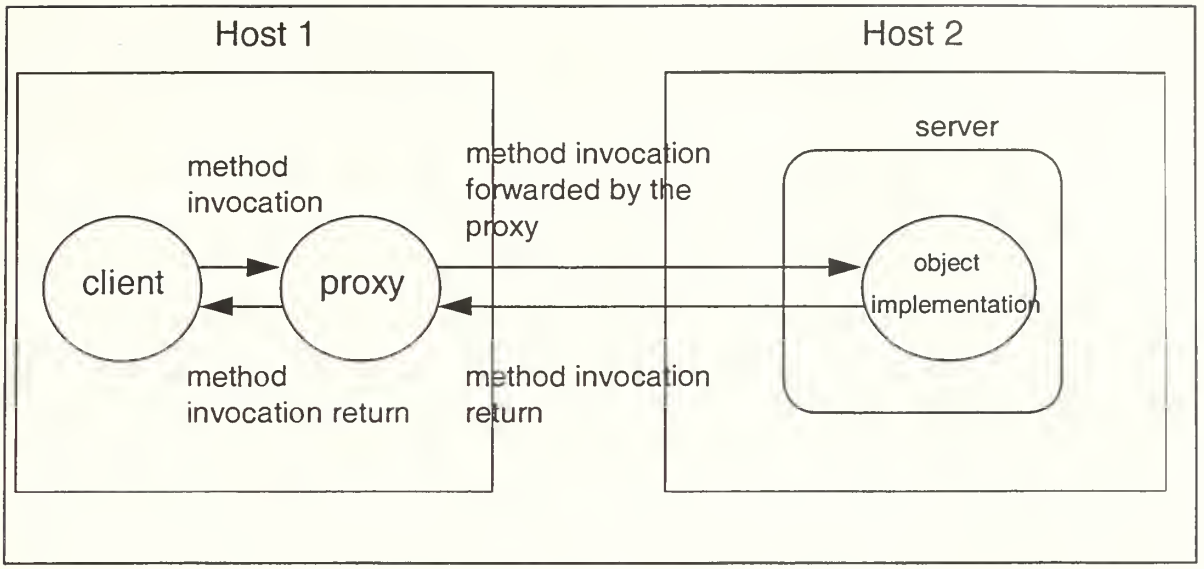


Figure 2.5 The Operation Call on a Proxy

The Interface Repository makes dynamic invocation possible. The Interface Repository maintains full information about interface and type definitions of the IDL definitions and provides this information at run time. Given an object reference, we can obtain the interface and all the information about the interface at runtime by calling Interface Repository's API's.

4. Object Implementation

ORB servers, also called object implementations to distinguish them from the interface of objects, provide the actual state and behavior of an object. The ORB core (Figure 2.2) locates an object adapter and forwards requests to the object implementations through the skeletons. The skeletons are the server IDL stubs that implement the services that the object exports. In addition, CORBA 2.0 also introduces dynamic skeleton to provide a run-time binding mechanism for server objects.

The object adapter makes use of the ORB's core communication services to accept requests on behalf of server objects. It provides the run-time environment to activate server objects, pass requests to them, assign them an object reference and ensure the security of interactions. It also records the classes it supports and their run-time instances in the Implementation Repository. (Figure 2.2)

The Implementation Repository maintains information that allows an ORB to locate and activate implementations of objects. Therefore, the server programmers must register their code with to the Implementation Repository.

5. ORB and RPC

Anyone experienced with writing client/server applications using RPC (Remote Procedure Call), will find that the mechanisms of an ORB and RPC are somewhat similar. Both must specify the interface first (object and its methods in ORB, but remote procedure names in RPC), use a compiler to generate both client and server stubs, then link the stubs with your client/server programs. However, there are some significant differences between ORB and RPC. With RPC, we call a specific function by its name and the data is separate from the function. In contrast, with the ORB we can invoke a method within a specific object which manipulates its own private data. The advantages of ORB over RPC are basically the same as the advantages of object-oriented over procedural programming language. Furthermore, CORBA provides not only the communication mechanism between clients and servers, but also a complete environment to create portable and interoperable client/server applications.

6. Programming Environment

This thesis was developed on the JavaORB for Windows NT 4.0 operating system with the JavaORB compiler. This environment was selected for the following reasons: First, Java is a mature and stable product. Second, Java IDL supports CORBA 2.0 features which has a formal specification with IDL to Java mapping. Finally, Windows NT is more accessible than UNIX workstations and higher reliability than DOS/Windows 3.1/Windows 95. For cost consideration and future expansion, we chose Windows NT instead of UNIX.

III. SYSTEM OVERVIEW

This chapter introduces the transaction concept and describes a global view of our Object Transaction Service (OTS). The OTS is treated as a "black box" here, and the components that cooperate to provide the transaction service will be explored in the next chapter. We focus our attention on the relationship between transactional clients, servers and the transaction service. OTS operates as a transaction manager. Moreover, OTS also provides a mechanism to allow the participants to cooperate with the transaction manager to complete their work.

A. TRANSACTION DEFINITION

OTS provides transaction synchronization across the elements of a distributed client-server application. In a typical scenario, a client initiates a transaction and then issues requests. Eventually, the client decides to end the transaction. If there are no failures, changes are committed; otherwise, changes are rolled back.

B. TRANSACTION PROPERTIES

As introduced in chapter one, the transaction concept is essential for building reliable distributed applications. We define a transaction as a unit of work, which comprises several operations made on one or more shared system resources that are governed by ACID properties, where *ACID* stands for the **Atomicity**, **Consistency**, **Isolation**, and **Durability** properties of a transaction. A unit of work can be transactional only if it satisfies these properties. Transactions are sequences of operations that are clustered together.

1. Atomicity Property

The **Atomicity** property of a transaction requires that this cluster is either performed completely, i.e. every single operation belonging to the transaction is successfully executed, or not at all. There is no partial execution of the transaction. Let us consider the example of the course registration and assume that we perform it as a transaction.

Atomicity states that either both the adding and dropping course operation are executed together (meaning that the course is being successfully added to the Registrated Course Table) or none of the operation is performed (meaning that the course is left where it was). A state in which a course is lost cannot occur. The start of a transaction denotes a state to which the transaction rolls back if any of its operations fails. If the transaction is completed successfully, the end of the transaction (which is usually the start of the next transaction) marks the next valid state.

2. Consistency Property

The **Consistency** property of a transaction requires that the sequence of operations leave the set of shared resources in a consistent state at the end of the transaction. This does not imply that states of inconsistency never occur, but their occurrence is confined to within a transaction. As such, inconsistent states are hidden for concurrent transactions and they must be resolved before the transaction is completed.

Revisiting the example of the course registration, a consistency constraint would be that a course is not lost. This is true at the end of the transaction when the course that has been copied from one table to the other by adding the name of the student. Within the transaction, after the adding has been completed but before the dropping has taken place, the set of account objects are in an inconsistent state as the course iteration is not complete. Hence it is the application that defines the notion of consistency and it is also the application that is in charge of ensuring consistency maintenance. If the transaction has reached a certain state of inconsistency that cannot be resolved, it can abort itself and recover to the consistent state from which it started. Considering our course registration example again, if the course has been added to Registered Course table but the other course is not dropped, the transaction can abort and the transaction mechanism will rollback to the previous valid state.

3. Isolation Property

The **Isolation** property of a transaction requires that the sequence of operations is performed in isolation from any concurrent transaction or unprotected activity. This means also, that any modifications that a transaction makes are not visible to other resources before the end of the transaction.

In this way other transactions or unprotected activities can never see an inconsistent state that may arise within a transaction. Likewise, operations performed within a transaction can never access modifications of other concurrent transactions.

4. Durability Property

The **Durability** property requires the effect of a transaction to be persistent so that it cannot be affected by failures. This requires a copy of all modified resources or a representation of the modifications (a change log) on persistent storage. This persistent representation can be consulted after a failure to recover to the state of the last successful transaction. In the case of a transaction between two register objects a transaction manager would either update a persistent representation of the two register objects at the end of the transaction, or it would add representations of the operation executions, i.e. the drop and the add operation with the actual parameter values, to a persistent log. Although hard disks are most commonly used to achieve persistent storage, this need not necessarily be the case. It could also be a battery-backed or an erasable PROM. A transaction can either terminate normally, or when some errors occur. When all the transactional operations can be performed successfully, the transaction commits and all the updates will be stored in the persistent storage. If any error occurs during the transaction processing, the transaction roll backs, that is, it terminates without any update to the data involved in the transaction. We observe that in order to support the requirement for atomicity, the data involved in a transaction must be recoverable. When a server object fails unexpectedly due to some hardware or software errors, the server must have the ability to recover itself from the persistent storage to retain the all-or-nothing property.

Transactions also can be nested. A nested transaction allows programs to have transactions embedded in existing transactions. Nested transactions have the advantage of providing finer granularity of recovery than flat transactions, but also require additional controls to transaction processing.

In our transaction service, objects involved in a transaction can play one of three roles: the *transactional client, transactional server* or *recoverable server.* We will examine each of these in the following sections.

C. TRANSACTIONAL CLIENT

A transactional client (TC), also called a transaction originator, is a process that begins a transaction(Figure 3.1). A transactional client first binds to OTS (Object Transaction Service) by means of ORB, and issues a *begin* call to OTS to create a new transaction context associated with the client program. Normally, the client then issues a set of transactional operations to several transactional objects which contain recoverable states. Recoverable, meaning that the objects participating in the transaction must provide uniform facilities that can cooperate with OTS to maintain the transaction properties described in section 3.B. Each transactional operation invoked on the target objects will be explicitly associated with a transaction context shared by all participants of the transaction.

The transactional client is oblivious to all the processing between OTS and server objects. It just begins a transaction and sends transactional requests to server objects, then commits or roll back the transaction.

D. TRANSACTIONAL SERVER

A transactional object is an object whose behavior is influenced by being invoked within the scope of a transaction. It is not necessary that all methods of a transactional object be transactional. A transactional object can provide both transactional and nontransactional operations. In contrast, an object that contains no transactional methods is a nontransactional object. Typically, a transactional object contains or indirectly refers to persistent data that can be modified by requests.

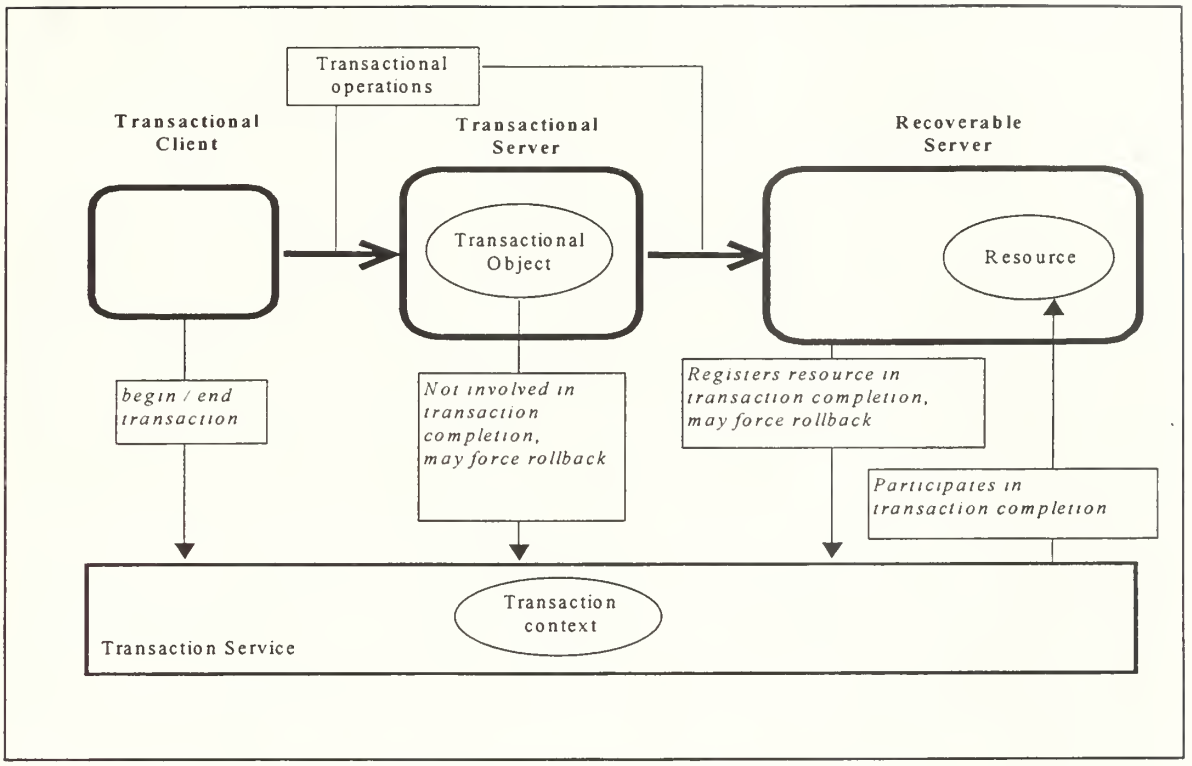


Figure 3.1 Object Transaction Service Overview

The Transaction Object receives requests for transactional operations from the Transactional Client and these requests may be forwarded to the Recoverable Objects or be performed by the Transactional Object itself.

The transactional server (Figure 3.1) is a collection of objects that are influenced by a transaction but has no recoverable states of its own. A transactional server propagates the transaction context to recoverable objects whose methods are invoked. In a real application, the transactional server may not exist. It is common for the recoverable server (Section 3.D) to be directly involved in the transaction and manage its own persistent data.

E. RECOVERABLE SERVER

A recoverable object contains persistent data and these data will be modified within a transaction. We can infer from the definition that a recoverable object is also a transactional object. The recoverable server (Figure 3.1) is a collection of objects and at least one of which is recoverable.

To participate in the completion of a transaction, a recoverable object has the responsibility for implementing some functions defined by OTS. With the help of these functions, an object can cooperate with OTS to ensure all participants have the same outcome(commit or rollback) and can recover themselves in the event of a failure. When recoverable servers are invoked by a transactional client for the first time, they will register themselves with OTS to become participants of the transactions. At the end of the transaction, recoverable objects are also involved in the two-phase commit protocol coordinated by OTS. During the processing of the transaction, recoverable objects must store certain information in the persistent storage at critical times. As a result, when a recoverable server restarts after a failure, it can recover its state and participate in a recovery protocol to complete the transaction.

OTS provides an architected set of IDL-ized interfaces for the objects that make up the transaction. The principal OTS module is defined to be *CosTransactions,* inside of which are defined nine key interfaces. In addition, for interoperability purposes, the OMG specified the *CosTSPortability* module separately.

The *CosTSInteroperation* module defines what is notionally referred to as a *transaction context*, a way for the OTS to keep track of the state of a particular part or thread of a transaction. This context contains the in-depth knowledge about what has transpired so far in a particular thread (in a threaded environment) or part of a transaction.

This is part of the mechanism used by CORBA transactional systems to impart the ACID properties that are the hallmark of a transaction. These objects cooperate to guide the principles of transaction processing. In addition to methods that manipulate their own data, the recoverable servers, by inheriting some of these interfaces, provide functions to participate in the transaction completion protocol.

F. OTS MANAGER

The OTS manager is the core of the object transaction service. It plays the role of coordinator among the transaction participants. In other words, the OTS manager cooperates with transaction participants to preserve the ACID properties during the execution of a transaction. OTS manager is composed of the following functional components: *Current*, *TransactionObject*, *TransactionFactory*, *Resource*, *Control*, *Terminator*, *Coordinator and RecoveryCoordinator*. These components are specified with IDL through which other objects may access their services via ORB (see Appendix A). The functionality of each of these components is illustrated via the following example. Figure 3-2 depicts the scenario that a TC cooperates with various functional components of the OTS manager to complete a transaction.

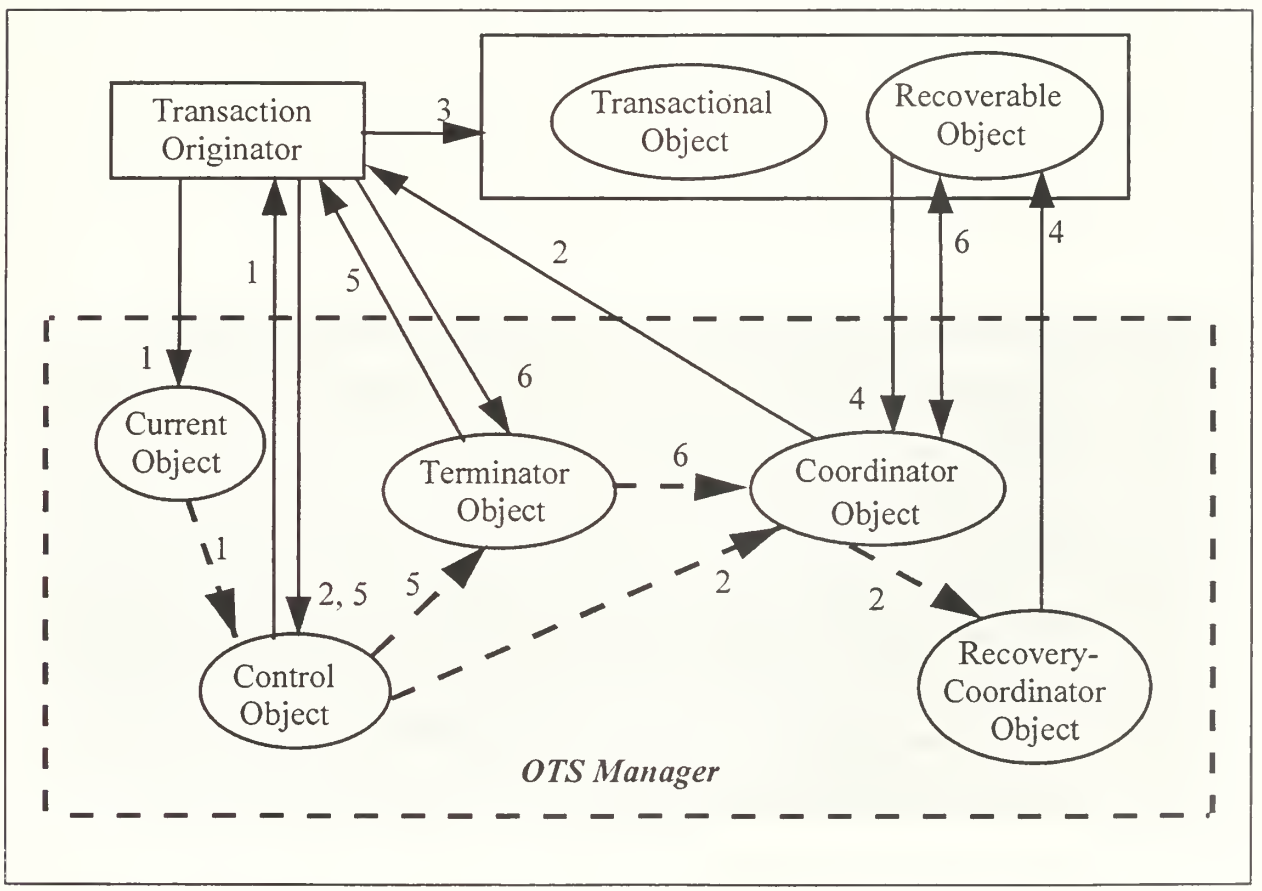


Figure 3.2. The Functional Diagram of the Applications and the OTS Manager

1. The transaction originator begins a new transaction by issuing a request to the Current object, and a unique Control object is returned.

```
org.omg.CORBA.Object obj =
        orb.resolve_initial_references("TransactionCurrent");
org.omg.CosTransactions.Current current =
                org.omg.CosTransactions.CurrentHelper.narrow( obj );
Control control = current.get_control();
```

2. Through the Control object, the client can connect to Coordinator in order to use transactional services.

```
Coordinator coordinator = control.get_coordinator();
```

3. The transaction originator then begins to invoke operations on the recoverable objects providing the Coordinator as an input parameter. *Register_resource* operation registers the specified resource as a participant in the transaction associated with the target object. When the transaction is terminated, the resource will receive requests to commit or rollback the updates performed as part of the transaction. These requests are described in the description of the Resource interface.

```
RecoveryCoordinator recCoordinator = coordinator.register_resource(r );
```

4. Recoverable objects will register their Resource objects with the Coordinator the first time they are invoked within the transaction.

```
myResource r = new myResource();
orb.connect( r );
```

5. The client uses the Control object to get the Terminator object to end the transaction.

```
current.begin();
current.commit(); or current.rollback();
```

The Coordinator will coordinate the termination process among Resource objects using a proper commit protocol.

```
coordinator.rollback_only();
```

G. TRANSACTION MODELS

The Transaction Service supports two distributed transaction models: flat transactions and nested transactions. A flat transaction is a transaction that cannot have child transactions. Nested transactions, on the other hand, allow subtransactions embedded within the current transaction.

Nested transactions provide a finer granularity of recovery than flat transactions. That is, when one of the subtransactions of a parent transaction fails, only the subtransaction is rolled back. The parent transaction has the opportunity to correct or compensate for the failure and finish its work. Moreover, subtransactions can be executed in parallel without the risk of inconsistent results.

A subtransaction is similar to a top-level transaction in that the changes made on behalf of a subtransaction are either committed in their entirety or rolled back. However, when a subtransaction is committed, the changes remain contingent upon commitment of all of the transaction's ancestors. A transaction cannot commit unless all of its children are completed. When a transaction is rolled back, all of its children are rolled back.

H. PROGRAMMING MODELS

To begin, the developer decides whether to use an *explicit* or *implicit* transaction context propagation model when writing the IDL for the CORBA objects. In the world of OTS, the scope of a transaction is defined by a transaction context that is shared by participating objects. A transaction context is created and becomes part of the environment of a transaction as the transaction begins. It spans the domain of the transaction by propagation among objects. In the programmer's point of view, propagation of transaction context in OTS can be implicit or explicit.

1. Implicit Method

Implicit propagation means that requests are implicitly associated with the client's transaction; they share the client's transaction context. The context is transmitted implicitly to the objects, without direct client intervention.

Implicit method requires the application domain CORBA object interface to inherit from the *TransactionalObject* interface. For example:

```
interface S1 :
        CosTransactions::TransactionalObject
        {
          void op1( void );
        }
```

This method does not require any modifications to the operation signatures, and the transactional context passes from point to point in the transaction 'invisibly' through the stub code. The inheritance requirement indicates to the OTS that transaction context information is contained in any IIOP messages that is exchanged between clients and servers.

2. Explicit Method

Explicit propagation means that an application object propagates a transaction context by passing objects defined by the Transaction Service as explicit parameters.

Explicit method requires that any interface operations that need to be transactional have an *in* parameter of type *Control*. In essence, this is a way for the transaction to pass itself along from point to point (or server to server) in the transaction because *Control* is a sort of handle to the transaction itself. An example would be:

```
interface S1
        {
                void op1(  in CosTransactions::Control x );
        }
```

IV. SYSTEM ARCHITECTURE

In Chapter III, we treat Object Transaction Service (OTS) as a "black box" and focus on the discussion between OTS and transactional clients and servers. In this chapter, we look into the "black box" and explore our system architecture.

A. OVERVIEW

Object Transaction Service (OTS) defines a set of IDL-ized interfaces for objects that make up the transaction service. Each IDL interface defines a functional component and OTS uses these components. Figure 4.1 illustrates the system architecture of OTS.

In the general scenario, the transaction originator begins a new transaction by issuing a request to a *TransactionFactory* object and a *Control* object is returned. With the methods provided by the *Control* object, the transaction originator can get a *Terminator* object and a *Coordinator* object.

```
interface TransactionFactory {
        Control create(in unsigned long time_out);
};
interface Control {
        Terminator get_terminator() raises (Unavailable);
        Coordinator get_coordinator() raises (Unavailable);
};
```

The transaction originator can use a *Terminator* object to commit the transaction after finishing all its transactional operations or rollback the transaction directly.

The *Coordinator* object is made available to the recoverable objects by associating it with the transaction context that is explicitly passed to recoverable objects as a parameter of each transactional operation. When a recoverable object is invoked within a transaction scope for the first time, it registers a *Resource* object with the *Coordinator* object to participate in the transaction.

```
myResource r  =  new myResource();
orb.connect ( r );
org.omg.CosTransactions.Current current  =  getCurrent();
Control control  =  current.get_control();
Coordinator coordinator  =  control.get_coordinator();
RecoveryCoordinator  recCoordinator  =  coordinator.register_resource(r );
```

A *Resource* object implements the two-phase commit protocol that is derived by the *Coordinator* object. In some failure cases, a recoverable object can contact a *RecoveryCoordinator* object to determine the outcome of the transaction and complete the transaction on its side. In the next section, we will take a close look at these components that make up OTS.

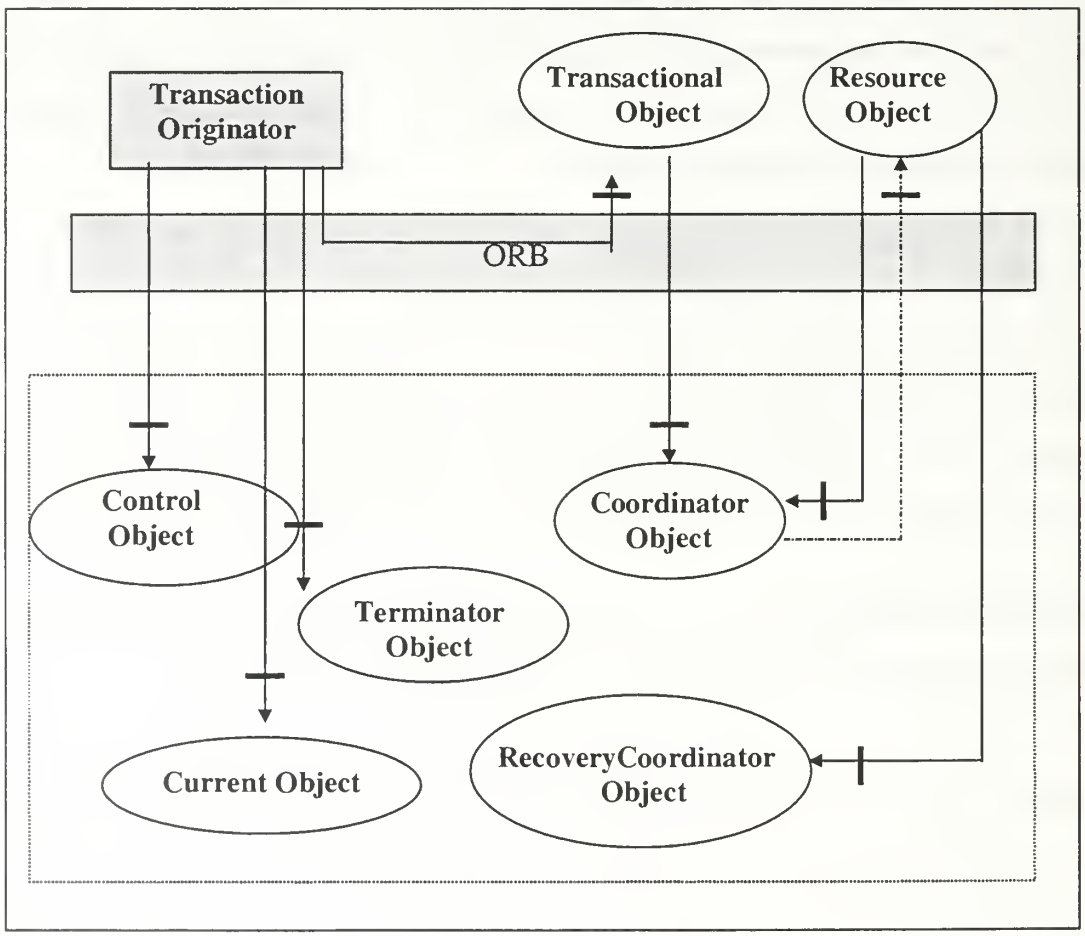


Figure 4.1 System Architecture of OTS

B. OBJECT TRANSACTION SERVICE INTERFACES

We describe the interfaces of OTS in this section. Each of these interfaces is specified by Interface Definition Language(IDL) and represents a functional unit of the transaction service. Each interface defines the operations and parameters that are exported to other components of the service. The OTS interfaces specified by the OMG are listed in Appendix A, and their descriptions also can be found in [1]. These interfaces will be introduced by following the programming logic. Full details of every operation will be presented. The transaction coordinator as we have introduced it above is specified through two interfaces: *Current* and *Coordinator*.

The *Current* interface specifies operations through which transactional clients access transaction operations to start, commit or abort a transaction. The *Coordinator* interface specifies coordination operations that are used by transactional servers (resources in the CORBA terminology). Finally, the *Resource* interface specifies operations that are to be provided by transactional servers in order to implement the two-phase commit protocol.

1. Current Interface

```
interface Current {
        void begin() raises(SubtransactionsUnavailable);
        void commit(in boolean report_heuristics) raises (NoTransaction, HeuristicMixed,
                                                          HeuristicHazard);
        void rollback() raises(NoTransaction);
        Status get_status();
        string get_transaction_name();
        void set_timeout(in unsigned long seconds);
        Control get_control();
        Control suspend();
        void resume(in Control which) raises(InvalidControl);
};
```

Interface Current defines the operations that transactional clients use to manipulate transactions. Operation *begin* starts a new transaction. If a transaction has been started before, it starts a subtransaction of the previously started transaction.

Operation *commit* ends a (sub) transaction. If the transaction is the root transaction, all changes are made durable. Operation *rollback* reverts all resources participating in the (sub)transaction to the state they had when the transaction started. The operation *get_control* provides a reference to the CORBA object that is the transactional server´s interface. Operation *suspend* stops the execution of a transaction (without aborting it). It returns a reference to a coordinator which can be used as an argument to the resume operation, which continues the execution of that transaction.

2. Coordinator Interface

```
interface Coordinator {
        Status get_status();
        Status get_parent_status();
        Status get_top_level_status();
        boolean is_same_transaction(in Coordinator tr);
        boolean is_related_transaction(in Coordinator tr);
        RecoveryCoordinator register_resource(
                                in Resource r) raises(Inactive);
        void register_subtran_aware(
                                in SubtransactionAwareResource r)
                                raises(Inactive, NotSubtransaction);
        ...
};
```

This Interface presents an excerpt of the Coordinator interface, which is also part of the transactional coordinator and is used by transactional servers. The two most important operations of the coordinator interface are *register_resource* and *register_subtran_aware*. They add a transactional server (i.e. a resource) that is passed as an argument to the set of resources that participate in a transaction or a subtransaction. The other operations can be used by transactional servers to obtain status information about the transaction, its parent or the root transaction.

3. Resource Interface

```
interface Resource {
        Vote prepare();
        void rollback() raises(...);
        void commit() raises(...);
        void commit_one_phase raises(...);
        void forget();
};
interface SubtransactionAwareResource:Resource {
        void commit_subtransaction(in Coordinator p);
        void rollback_subtransaction();
};
```

Resources that participate in flat transactions must be defined as subtypes of Resource and those transactional servers that participate in subtransactions must be subtypes of SubtransactionAwareResource. They then have to redefine the operations *prepare*, *rollback*, *commit*, *commit_one_phase* and *forget*. In the implementation of these operations, transactional servers would then implement their part of the two phase commit protocol.

C. CLIENT AND SERVERS

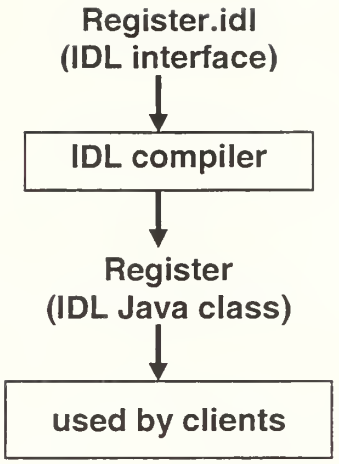
An IDL interface looks like the definition of a C++ class and it indeed maps to an actual Java class (certainly, it can be any other object-oriented language in other CORBA implementations). Every object that wishes to be accessible across the ORB must specify its IDL interface at first, then an IDL compiler generates the mapping Java class, called IDL Java class, according to the IDL interface in Figure 4.2, it defines *Register* interface. An IDL Java class lists the functions that clients of the interface can invoke, and these functions must be defined in Java (or other object-oriented language) by the implementer of the interface. With the help of the IDL Java class, a client can access the remote implementation objects in the CORBA environment.

A single Java object can implement multiple interfaces and an IDL interface also can have different server objects to implement it in different ways.

An object server can contain any number of server objects that implement the same or different IDL interfaces. An object must register itself with an ORB before any client can access it. A registered server will be activated by an ORB if it is dormant when one of its methods is invoked.

```
#include "CosTransactions.idl"

module Project{
        exception UnavailableCourse
        {
                string course_name;
        };
        struct course
        {
                string index_number;
                string course_code;
                string section;
                string course_name;
                string meeting;
                string credits;
                string student_ID;
        };
        typedef sequence<course>courseSeq;

        interface Register :  CosTransactions::TransactionalObject
        {
            courseSeq add_course(in string course_index_number,
                                 in string Student_ID, in   string password ,
                                  in string studentName)
                            raises ( UnavailableCourse );
           courseSeq drop_course(in string course_index_number, in string Student_ID,
                                          in string password,  in string studentName)
                            raises ( UnavailableCourse );
        };
};
```

(a)



(b)

Figure 4.2 (a) The Register IDL Interface
(b) The Programming Environment for Clients

1. Begin a Transaction

Initially, there may be one or more OTS servers available in a distributed environment. Each of these OTS servers provides the same *Current* object. A transaction originator can choose any OTS server to be the coordinator site of its transaction. A transaction originator first binds to the *TransactionCurrent* object, then invokes the *begin()* operation of *Current* to begin a new transaction.

```
org.omg.CosTransactions.Current current = getCurrent();
current.begin();
-----------------------------------------------------------------
private org.omg.CosTransactions.Current getCurrent() {
        System.err.println("Session: resolve transaction current");
         try{
                org.omg.CORBA.Object    obj   =
                orb.resolve_initial_references("TransactionCurrent");
                 org.omg.CosTransactions.Current current =
                 org.omg.CosTransactions.CurrentHelper.narrow( obj );
```

```
            if( current == null ) {
                System.err.println("current is not of expected type");
            }
            return current;
        }//end of try block
        catch( org.omg.CORBA.ORBPackage.InvalidName in )
        {
            System.err.println( in );
            return null;
        }//end of catch block
        }//end of getCurrent() method
```

When receiving the request, the *Current* object will create a *Control* object and return a reference to the *Control* object to the client (Figure 4.3). The transaction originator also can give a time-out value to set the maximum transaction processing time. If the time-out expires, the transaction will rollback.

```
void set_timeout(in unsigned long seconds);
```

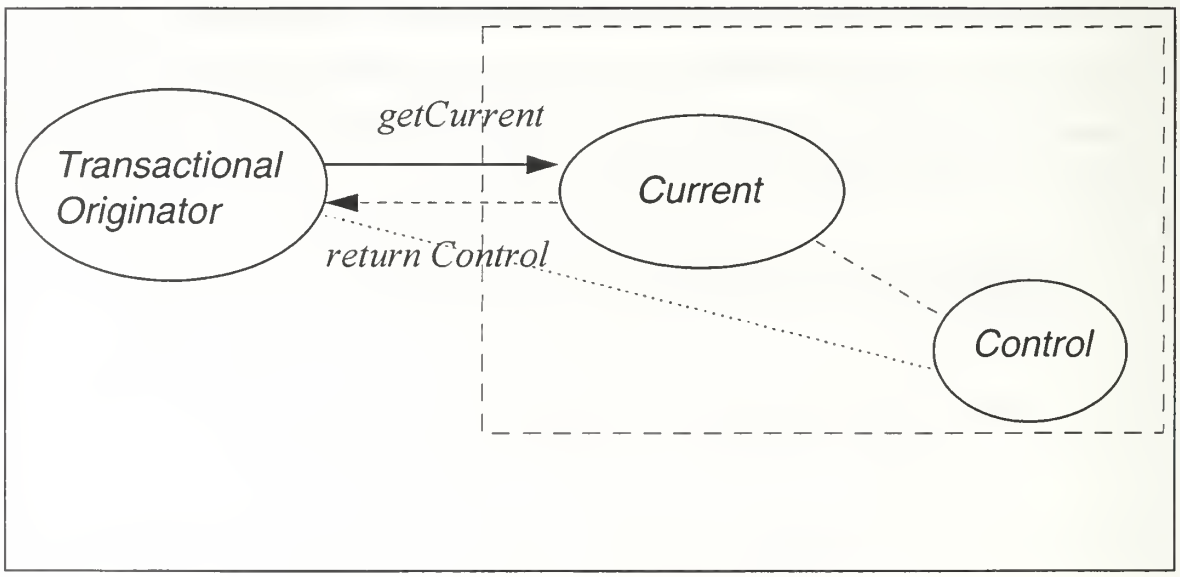


Figure 4.3 Begin a Transaction

After creating the Control object to control the new transaction, The Control object is associated with the new transaction and responsible for the processing of the transaction.

2. Transaction Coordinator

When a *Control* object is created, a *Terminator* object and a *Coordinator* object are also created. The *Control* object provides two functions:

```
Terminator get_terminator() raises (Unavailable);
Coordinator get_coordinator() raises (Unavailable);
```

The *get_terminator()* operation of *Control* returns the object reference of the *Terminator* object and *get_coordinator()* operation *Control* returns the object reference of the *Coordinator* object. A transaction can involve multiple objects performing multiple requests. All participating objects share the same transaction context that defines the scope of the transaction. We must have a mechanism that allows the objects involved in the transaction service and add related information to the transaction context. A *Coordinator* object is responsible for maintaining the transaction context and providing operations that are used by participants of the transaction. A transaction participant, i.e., a recoverable object, must issue a *register_resource()* operation to the *Coordinator* object when one of its transactional operations is invoked by the transaction originator for the first time. Owing to we adopt implicit way to propagate the transaction context, just because of this reason I defined ,prepared a new *myResourse.java* class; all transactional operations of a recoverable object must have an object reference to the *Coordinator* object as the last parameter(Figure 4.4).

```
myResource r = new myResource();
orb.connect( r );
RecoveryCoordinator     recCoordinator   =   coordinator.register_resource(r );

-------------------------------------------------------------

public class myResource extends org.omg.CosTransactions._ResourceImplBase
                                        implements  org.omg.CosTransactions.Resource
{
        public org.omg.CosTransactions.Vote prepare()
        {
                System.out.println("Resource : PREPARE");
                // We indicate that we are OK to commit
                return org.omg.CosTransactions.Vote.VoteCommit;
        }//end of prepare method
```

```
        / * Operation rollback*/
        public void rollback(){
                System.out.println("Resource : ROLLBACK");
        } //end of rollback method
        /* Operation commit          */
        public void commit()
        {
            System.out.println("Resource : COMMIT");
        }//end of commit method
        / * Operation commit_one_phase     */
        public void commit_one_phase()     {

            System.out.println("Resource : COMMIT_ONE_PHASE");

        }//end of commit_one_phase method
        / * Operation forget */
        public void forget()
        {
            System.out.println("Resource : FORGET");
        }//end of forget method
}//end of myResource class.
```

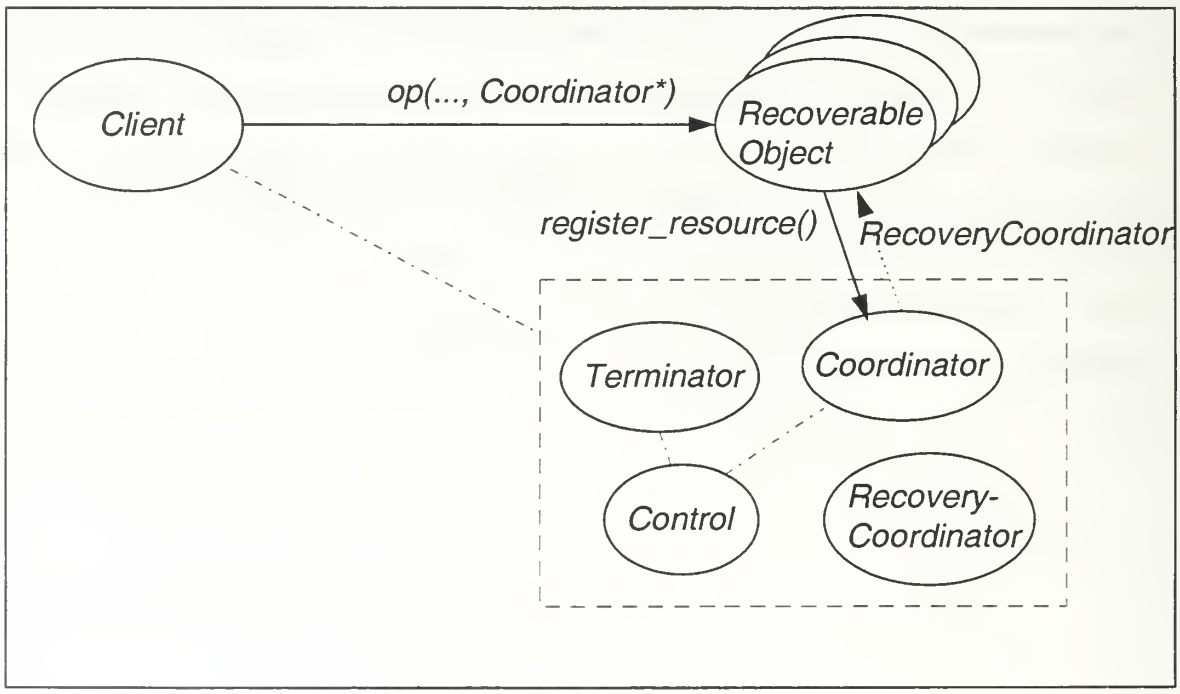


Figure 4.4 The Propagation of Transaction Context

A *Coordinator* object maintains a database that records the object references of the transaction participants. When the transactional originator commits the transaction, the *Coordinator* object uses this information to perform the two-phase commit.

The *Terminator* object defines two operations to end a transaction. Typically, these operations are used by the transaction originator. The transaction originator issues the *commit()* operation to mark the end of the transaction. In some cases, a transaction originator may wish to rollback the transaction it creates directly, then it uses a *rollback()* operation to end the transaction. The *register_resource()* operation of *Coordinator* returns the object reference of a *RecoveryCoordinator* to the recoverable object.

```
RecoveryCoordinator  recCoordinator  =  coordinator.register_resource(this );
```

In certain situations, a recoverable object can use the *replay_completion()* function provided by *RecoveryCoordinator* object to drive the recovery process(Figure 4.4).

3. Transaction Participants

Except for their own operations, transaction participants, i.e., recoverable objects, must implement transactional behavior to ensure the isolation and durability properties of transactions. To complete a transaction, transaction participants have the responsibilities to:

- register themselves with the Coordinator object for transaction completion;
- participate in two phase commit protocol; and
- support transaction recovery.

While registration was described in Section 4.2.3 and issues about recovery will be left to the next chapter, we discuss the two-phase commit protocol in this section.

The Object Transaction Service uses the two-phase commit protocol to complete a transaction with each registered Resource object. The two-phase commit protocol is a well known atomic commit protocol and is applied in many practical systems.

It is designed to allow any recoverable object to abort its part of a transaction. If any part of a transaction is aborted, then the whole transaction must be aborted for atomicity.

As implied in the name, the processing of the two-phase commit protocol can be divided into two phases. In the first phase, all recoverable objects(transaction participants) return their votes for the transaction outcome. In the second phase, every transaction participant carries out the joint decision.

It is the transaction originator that issues commit (or rollback) to the transaction and the request is directed to the *Coordinator* object. The *Coordinator* object then communicates with the transaction participants to complete the two-phase commit protocol. The *Resource* object provides operations invoked by the *Coordinator* object on each recoverable object. The communication of two-phase commit protocol is depicted in Figure 4.5.

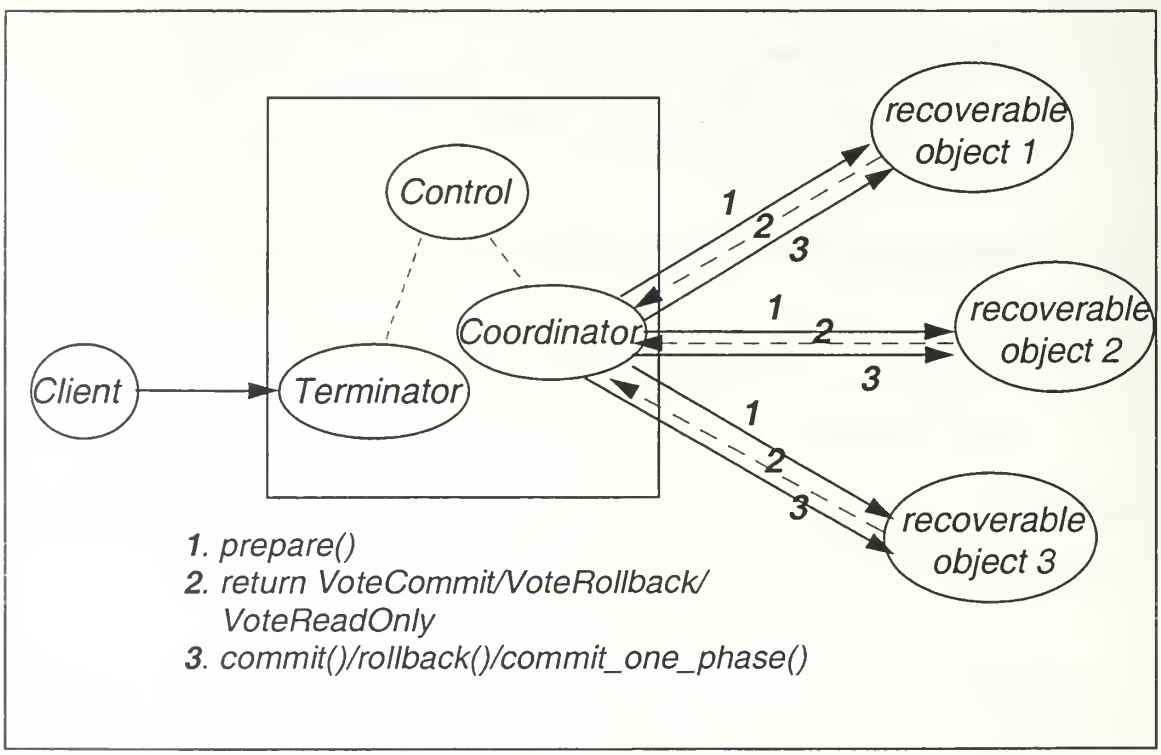


Figure 4.5 Communication in Two-Phase Commit Protocol

4. Processing of Two-Phase Commit Protocol

The processing of two-phase commit protocol can be divided into the following four steps:

1. The *Coordinator* object invokes the *prepare()* method on each recoverable object.

2. When a recoverable object's *prepare()* is invoked, it checks its own state to see if it can commit its part of the transaction, then replies its vote to the *Coordinator* object. The vote can be *VoteReadOnly, VoteCommit* or *VoteRollback.*

3. The *Coordinator* object collects the votes from each recoverable object, then (a) If any recoverable object returns *VoteRollback*, the *Coordinator* object decides to rollback the transaction and invokes the *rollback()* on all recoverable objects which reply *VoteCommit*. (b) If at least one recoverable object votes *VoteCommit* and all others vote *VoteCommit* or *VoteReadOnly*, the *Coordinator* object can commit the transaction by invoking the *commit()* on each of recoverable objects. (c) If all recoverable objects vote *VoteReadOnly*, the transaction can complete immediately and there is no further operation is required.

4. Recoverable objects that vote *VoteCommit* are waiting for a *commit()* or a *rollback()* from the *Coordinator* object. Each of the recoverable objects must implement their commit and rollback operations, so that they can act accordingly.

```
interface Resource {
        Vote prepare();
        void rollback() raises(...);
        void commit() raises (...);
        void commit_one_phase() raises(..);
        void forget();
};
```

In the special case of only one participant registered for a transaction, the first phase(voting phase) is not necessary. Instead of issuing *prepare ()* and *commit()* or *rollback()* on the single recoverable object, the *Coordinator* object can invoke *commit_one_phase()*.

V. IMPLEMENTATION ISSUES

A. BACKGROUND REVIEW

As we have discussed in Chapter IV, when a transactional client begins a new transaction, a set of functional components, including *Control*, *Terminator*, *Coordinator* and *RecoveryCoordinator*, will be created to play the manager role of the new transaction. OTS provides separate IDL interfaces that define the above components. The benefit of dividing the transaction manager role into several functional components is that OTS can create several different views of the transaction manager. The transactional client and recoverable servers can only access the indispensable part of manager functions. For example, if OTS passes the reference of a *Coordinator* object to the recoverable servers, then the recoverable servers have no access to the operations defined by the *Terminator* interface to end the transaction.

As described before, an object can implement multiple IDL interfaces if it provides all the functions in these interfaces. For the convenience of implementation, we construct a composite object to implement these four interfaces. But the users still only can get the separate views of the transaction manager. As shown in Figure 5.1, the IDL compiler generates an IDL Java class for each interface and Java ORB provides mechanisms to "tie" the IDL Java class and its object implementation together. The programming environment hides the actual implementation from the users (transactional client and recoverable servers), and creates separate views for them.

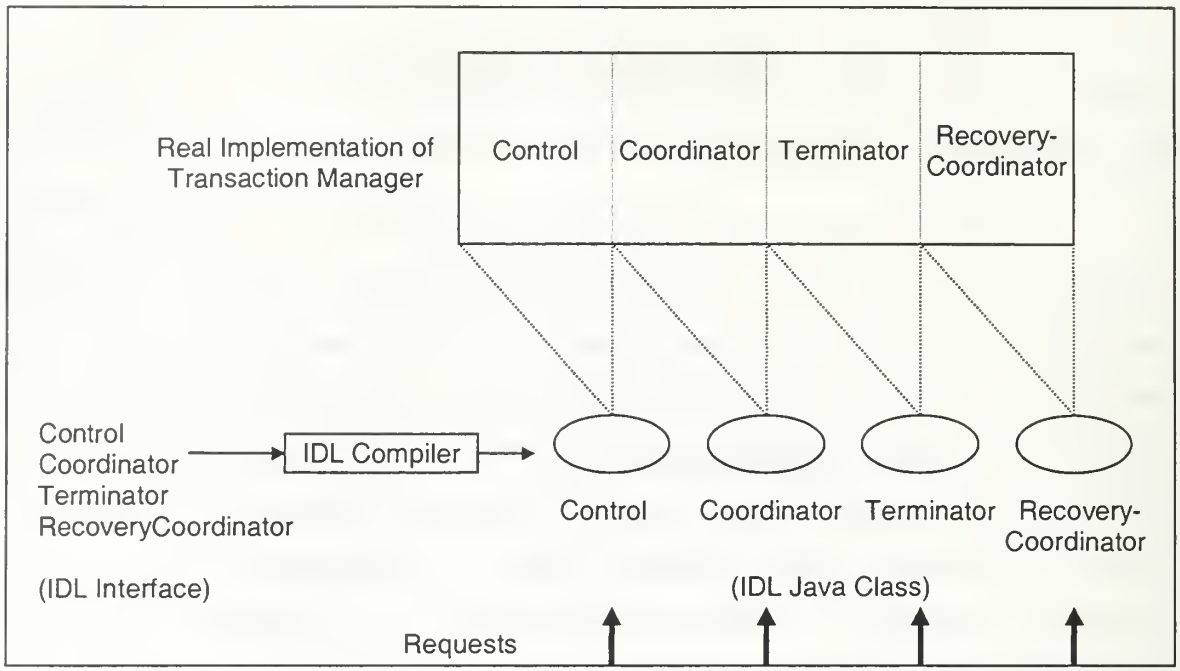


Figure 5.1 The Interfaces and Transaction Manager

B. OVERVIEW OF THE EXAMPLE

This chapter describes the development of distributed, object-based transactional applications with JavaORB OTS using a sample Java application. At the beginning of each quarter all Computer Science students fill a course request form for the next quarter course schedule. The iteration scenario requires students to access a remote database to insert or delete a class from *Registered_Course* table. Students select the *Course* from the course table. Each student accesses database by using a Client program defined in Java. A server program controls access to the database.

Computer Science students need to make some transactions on this database. Using Corba we can create a distributed, client /server, object-based transactional program.

What happens if any error occurs during a single user's transaction? For example, attempting to register for an invalid course, or attempting to drop a course for which the student is not registered. How can we turn back without making any change to database? Corba Object Transaction Service will handle this problem.

What happens when two or more users attempt to update a *Registered_Course* table at the same time? For this purpose, *synchronized* methods and locks can be used to solve concurrency problem.

During a transaction, a course is transferred between the *Course* and the *Registered_Course* tables-depending upon the parameters passed to the client program.

The programs for the Registration Request are:

1. RegistrationImpl : The *RegistrationImpl* program takes input from the Client program via a Server object. This program connects to the database by using JDBC. As a database application this programs takes the desired course from the student and copies this course from the Core Course tables and inserts this course in the *Registered_Course* table together with the name of the student. It then begins a transaction and performs the requested transfer. After all requested transfers have been completed, it requests to complete the transaction (either commit or rollback). A lock object is used in both synchronized *drop_course* and *add_course* methods to lock these methods for concurrency transactions, also unlock function is activated in commit or rollback statements.

2. Server : The Server program, creates a Server object and binds to a *RegistrationImpl* object .

3. Client : The Client Program prepares the GUI to assist the user student in entering the new course for adding or for dropping an old course from the database. Also the client program displays the current course schedule. The client ensures that changes made during the transaction to courses are stored persistently (if committed), or that the courses are returned to their state before the transaction (if rolled back).

Development and design of a sample CORBA project, is discussed below. Unlike the other examples, this is a more complete one involving a GUI interface and a backend database. JavaORB from Distributed Object Group (DOG). is used because this vendor supplies a free version of Corba Transaction Service together with CORBA architecture.

C. A JAVA – CORBA SAMPLE

Here are the steps to implement The Course Registration Program:

1 . Implement a simple interface in IDL that defines the Register object required for the Course Registration application, See "Writing the Project IDL" in this chapter.

2 . Implement the client program and transaction originator (Client): gather input from the user about which course will be added or dropped from the database, initialize the ORB, get the Register Reference from a file, narrow the object reference to Register object, obtain a reference to a transactional object (Register) in *add_course* and drop_*course* methods, perform actions with the transactional object (Register), commit or rollback the transaction, and handle exceptions.

3 . Implement the Server program: initialize the ORB and BOA, create and register the object then declare the transactional object, connect the instance to the BOA, export the object reference into a file, then wait for incoming requests,

4 . Implement the *RegistrationImpl*: this class handles the database connectivity for the server side, also corresponding to the request of the client, make add and drop courses from the tables under the control of transaction rules defined by this itself, performs actions with the transactional object (Register), commit or rollback the transaction, and handle exceptions.

5 . Implement *myResource* class: defines the resource class operation that will be used *myresource* class used in *RegistrationImpl* class.

6 . Implement lock class: handles the lock and unlock function that is used in *add* and *drop course* methods.

D. WRITING THE PROJECT "IDL"

The first step to creating a transactional application with JavaORB OTS is to specify all of our interfaces using the CORBA Interface Definition language (IDL). IDL is language-independent and has a syntax similar to Java, but can be mapped to a variety of programming languages.

```
#include "CosTransactions.idl"

module Project{
        exception UnavailableCourse
        {
                string course_name;
        };
        struct course
        {
                string index_number;
                string course_code;
                string section;
                string course_name;
                string meeting;
                string credits;
                string student_ID;
        };
        typedef sequence<course>courseSeq;

        interface Register :  CosTransactions::TransactionalObject
        {
              courseSeq add_course(in string course_index_number,
                        in string Student_ID, in   string password ,
                    in string studentName)
                        raises ( UnavailableCourse );
             courseSeq drop_course(in string course_index_number, in string Student_ID,
                                in string password,  in string studentName)
        raises ( UnavailableCourse );
        };
};
```

Figure 5.2 Project IDL

IDL sample shows the contents of the Project.idl file which defines the three objects required for the Registration Course. Note that;

The Register interfaces inherits from *CosTransactions::TransactionalObject* because this interface must participate implicitly in the transaction. If we do not inherit it from *CosTransactions::TransactionalObject* it does not participate in the transaction. Also note that the IDL file must include the CosTransactions.idl file.

This provides the IDL for the *CosTransactions::TransactionalObject* interface from which the transactional objects must inherit. The IDL is used by the JavaORB's idl2java compiler to generate Java stub routines for the client program, and skeleton code for the server objects. The stub routines are used by the client program for all method invocations.

We use the skeleton code, along with code we write, to create the server programs that implement the objects. The code for the client and servers, once completed, is used as input to our Java compiler to produce your client and server programs.

E. INTERFACE DEFINITION LANGUAGE (IDL) TO JAVA MAPPING

When developing a CORBA application, the first step is write the IDL file(s) for the project. After developing the IDL, the idl2java compiler was used to create the necessary stubs and skeletons for Java, so that CORBA clients can communicate with our CORBA server. It should be noted that the IDL is part of the CORBA standard; therefore, it may be used with any CORBA implementation. IDL is used to specify CORBA objects, what methods they have, what types they return, etc. The example in Figure 5.2 is an idl file called *Project.idl*. It defines one CORBA object: Register. As we can see it does not resemble Java. IDL does not provide implementation but the interfaces to the object.

1. Modules

An interface can be defined within a module. This allows interfaces and other IDL-type definitions to be grouped together in a useful fashion. Modules also create naming scope. This means that a type name used in one module, will not conflict with the same name used in another module. The idl2java compiler, maps modules to Java packages.

2. Interfaces

An IDL interface provides a description of the functionality that will be provided by a CORBA object. An interface provides all the information needed to develop a client that can use this defined interface to interact with the object. In the above Project.IDL we have the Register interface. In an interface you typically declare: constants, types, exceptions, attributes, and operations.

3. Exceptions

The standard way of processing errors in CORBA is through exceptions. An IDL operation may raise an exception indicating that an error has occurred. There is an example of an exception in Figure 5.2. This example shows that an exception *UnavailableCourse* is raised when the course code that was set is invalid. In addition to supporting customized exceptions, CORBA defines a set of standard exceptions.

4. Structures

A struct data type allows related items to be grouped together in a useful fashion. An IDL struct maps to a final Java class that contains one instance variable for each structure field. The class name is the same as the IDL structure name.

5. Sequences

A sequence is a one dimensional array with two characteristics: A maximum size (which is fixed at compile time) and a maximum length (which is determined at run-time). A sequence is similar to a one dimensional array but it is not fixed length.

Example:

typedef sequence<long> myUnboundedArray;

or you can have a bounded sequence:

typedef sequence<long, 10> myBoundedArray;

6. Strings

The string type is implemented in a way similar to a sequence of char, which may be bounded or unbounded. Below is an example:

```
interface library
{
        //Bounded string
        attribute string mystring<12>
        //Unbounded string
        attribute string title
}
```

7. CORBA Parameters

Corba defines three parameter passing modes: in, out and inout. In parameters pass from client to server, out parameters pass from server to client, and inout parameters pass from both directions.

F. WRITING THE TRANSACTION ORIGINATOR (CLIENT PROGRAM)

The file named Client.java contains the implementation of the Java client program that is also the transaction originator. The Client program gathers input from the user and performs a single OTS-managed transaction

The Client program performs these steps:

1 . Initializes the ORB.

2. CosTransactions Service initialization

3 . Get the Transactional server reference from an object file.

4. Narrow the object reference.

5 . Get the current object and Begin a transaction.

a. Obtains a reference to the transactional objects

6. Invokes the add_course() and drop_course() methods on the RegisterImpl class objects for each course entered into to the client program. It prints out the current Course Schedule for each Student after the transaction.

7. Commits or rolls back the transaction.

1. Initializing the ORB

The first task the transaction originator needs to do is initialize the ORB, as shown in Code sample 5.1. The parameters args and null must be passed to the ORB, the BOA, and the OTS Transaction Service instance.

Code sample 5.1 Example of Initializing the ORB with Java

```
...
public static void main(String[] args) throws Exception
{
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
...
```

2. CosTransactions Service Initialization

The second task the transaction originator needs to do is initialize the CosTransactions, as shown in Code sample 5.2. The parameter orb must be passed to the *Boot.init()* method.

Code sample 5.2 Example of initializing the CosTransactions Service

```
...
public static void main(String[] args) throws Exception
{
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
   org.omg.CosTransactions.Boot.init( orb );
...
```

3. Get the Transactional Server Reference from an Object File

The third task the transaction originator needs to do is get the Transactional server reference from an object file , as shown in Code sample 5.3. The client application is very simple, to get Transactional server reference we use a file " ObjectId " to which the Server will write its reference.

Code sample 5.3 Get the Transactional Server Reference from an Object File

```
.....
org.omg.CORBA.Object obj = null;
        try{
                java.io.FileInputStream file = new java.io.FileInputStream ("ObjectId");
                java.io.DataInputStream myInput = new java.io.DataInputStream(file);
                String stringTarget = myInput.readLine();
                obj = orb.string_to_object(stringTarget);
        }
        catch ( java.io.IOException e ){
                System.out.println("Server reference not available...");
                System.exit(0);
        }
....
```

4. Narrow the Object Reference

The fourth task the transaction originator needs to do is narrow the object reference, as shown in Code sample 5.4 references to remote objects in CORBA use a helper class to retrieve a value from that object. A commonly used method is the helper narrow method, which ensures the object is cast correctly.

Code sample 5.4 Narrow the Object Reference

```
.....
static Register orblet = null;
        try{
                java.io.FileInputStream file = new java.io.FileInputStream ("ObjectId");
                java.io.DataInputStream myInput = new java.io.DataInputStream(file);
                String stringTarget = myInput.readLine();
                obj = orb.string_to_object(stringTarget);
        }
        catch ( java.io.IOException e ){
                System.out.println("Server reference not available...");
                System.exit(0);
        }
// Narrow the object reference
orblet = RegisterHelper.narrow(obj);
....
```

5. Get the Current Object and Begin a Transaction

Before beginning a transaction, you must obtain a transaction context. OTS-managed transactions are handled transparently to your application with Current--an object that maintains a unique transaction for each active thread. To use an ITS-managed transaction, you must obtain a reference to this Current object. The Current object is valid for the entire process under which you create it, and can be used in any thread.

Code sample 5.5 shows how to obtain an OTS-managed transaction. First an object reference is obtained for the TransactionCurrent object using the *resolve_initial_references()* method. The Current object returned from this method is then narrowed to the specific CosTransactions.Current object using the *narrow*() method.

Code sample 5.5 Get the Current Object

```
...

private org.omg.CosTransactions.Current getCurrent() {
    System.err.println("Resolve TransactionCurrent to get access to Current  object.");
    try {
            org.omg.CORBA.Object obj = orb.resolve_initial_reference s ("TransactionCurrent");
            org.omg.CosTransactions.Current current =
            org.omg.CosTransactions.CurrentHelper.narrow( obj );
            if( current == null ) {
                    System.err.println("current is not of expected type");
            }//end of if
    return current;
    }//end of try

    catch ( java.lang.Exception e )
    {
            System.err.println(e);
            return null;
    }//end of catch block
}//end of getCurrent()
```

To perform work that is managed by OTS, you must first begin a transaction using the Current interface's begin() method. Only one OTS-managed transaction can be active within a thread at a time. Code sample 5.6 shows the transaction originator beginning an OTS-managed transaction.

Code sample 5.6 Example of Beginning a Transaction

```
...
    // get current
    org.omg.CosTransactions.Current current = getCurrent();
    // Begin a transaction
    System.out.println("Begin a transaction.");
    current.begin();
...
```

6. Invoking Methods on the RegisterImpl class from Client Program

To invoke methods of the Register Interface, we have to define the object (orblet) by the name of the Register Interface as described in Code Sample 5.3. This object is used to invoke the methods (*add_course()* and *drop_course()*). The methods of the Register Interface can be invoked If the object of that interface (register) being gained by the help of that Interface(Register). Code sample 5.6 shows the details of invoking a method of the RegisterImpl class from the Client side(Transaction Originator).

Code sample 5.6 Invoking the *add_course()* and *drop_course()* Methods on the RegisterImpl Class

```
  ......
  static course[] data = new course[100];
.........
  public void addCourse(){
        try {
              // get current
              org.omg.CosTransactions.Current current = getCurrent();
               // Begin a transaction
              System.out.println("Begin a transaction.");
               current.begin();
               System.out.println(addCourseTextField.getText());
              String course_index_number =  addCourseTextField.getText();
              String studentName=  studentNameTextField.getText();
              data = orblet.add_course(course_index_number, student_ID, password,
                          studentName);
              System.out.println(" Commit the transaction ");
              current.commit(true);
        System.out.println("The transaction has been commited");
        }//end of try
        catch ( java.lang.Exception e ){
              System.err.println("No transaction - rollback:\n " + e );
              try {
                    System.out.println("Current Transaction is rolling back");
                    current.rollback();
              }//end of try
              catch( org.omg.CosTransactions.NoTransaction nt) {
                    System.err.println("\nNo transaction "   + nt );
                    System.exit( 1 );
              }//end of second catch
     }//end of first catch
  }
```

7. Committing or Rolling Back a Transaction

Once a transaction has begun, it must be committed or rolled back to complete the transaction. If an originator of an OTS-managed transaction does not complete the transaction, the OTS Transaction Service will rollback the transaction after a timeout period. However, it is important to commit or rollback transactions so that hung transactions do not consume system resources. Code sample 5.6 shows how the client program uses the commit variable to decide whether to commit or rollback the transaction. If the commit variable is true, the transaction is committed. If the commit variable is false, the transaction is rolled back. In Code sample 5.6, the false parameter sent to current.commit() means that heuristics will not be reported. Note that the test for whether to commit or roll back is contained within a finally clause of the previous try clause. This is to ensure that an unexpected exception does not bypass this code.

Code sample 5.7 Example of Committing or Rolling Back the Transaction

```
...
boolean commit = false;
try
{
  ...
        commit = true;
}
...
finally
{
 // Commit or roll back the transaction
        if (commit) {
                System.out.println("*** Committing transaction ***");
                  current.commit(false);
        }
        else  {
                System.out.println("*** Rolling back transaction ***");
                  current.rollback();
        }
}
...
```

H. WRITING THE SERVER PROGRAM

The Server program performs these steps in the main routine:

1. Initializes the ORB.
2. CosTransactions Service initialization.
3. Initializes the object adaptor(BOA).
4. Declare the transactional object.
5. Connect the transactional object to BOA.
6. Convert the transactional object 's reference to string.
7. Put the transactional object's reference into a file.
8. Then wait for incoming requests from the Clients.

1. Initializing the ORB , the CosTransactions and the BOA in the Server Program

Before instantiating the Register object, the main routine must make three calls--one to the ORB, the other to the Basic Object Adaptor (BOA) and the third to CosTransactions. After the ORB is initialized the CosTransactions service is initialized using the current orb as a parameter.The BOA is the interface between the object implementation and the ORB. The BOA allows your object to notify the ORB when it is ready to accept client requests and informs it when client requests are received. The programming details for this explanation are given in Code sample 5.8.

Code sample 5.8 Initializing the ORB, CosTransactions and BOA in the Server

```
public static void main(String args[]) throws Exception {
        // Initialize the ORB
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // CosTransactions Service initialization
        org.omg.CosTransactions.Boot.init( orb );
        // Initialize the BOA
        org.omg.CORBA.BOA boa = org.omg.CORBA.BOA.init(orb,args);
```

2. Declare the Transactional Object and Connect it to Basic Object Adaptor

While explaining the *Project.idl* in Figure 5.2, we focused on the *Register* interface. We pointed out that this interface is a Transactional object. For this reason we defined on the server side a new object by using *RegisterImpl* class (actually extends *_RegisterImplBase* that is also transactional) by sending *orb* object, to make a direct relation between Server class and RegisterImpl class. Then we connect the instance of this object to the basic object adapter that is initialized in this server program to make connectivity between Server and RegisterImpl efficient. The programming details for this explanation was given details in Code sample 5.9.

Code sample 5.9 Declare the Transactional Object and Connect it to Basic Object Adaptor

```
// Then declare the transactional object
RegistrationImpl orblet = new RegistrationImpl(orb);

// Connect the instance to BOA
boa.connect(orblet);
```

3. Convert the Transactional Object 's Reference to String and Convert it's Reference to String

On the client side a file named an "*ObjectID*" was used to read the object reference of the Server side to make a connection to the client side object reference via the basic object adapter.(Code sample 5.3) Next a string reference for the transactional Register Object is obtained and placed in a "*ObjectID*" file to make the client side connectivity by the help of basic object adaptor. The programming details for this explanation was given details in Code sample 5.10.

Code sample 5.10 Convert the transactional object 's reference to string and Convert it's reference to string

```
.....

// Export into a file the object reference
String reference = orb.object_to_string(orblet);


//then put it into a file
try{
        java.io.FileOutputStream file = new java.io.FileOutputStream("ObjectId");
        java.io.PrintStream pfile  =  new java.io.PrintStream(file);
        pfile.println(reference);
}//end of try

catch ( java.io.IOException ex ){
        System.out.println("Unable to export server reference");
}//end of catch

.....
```

4. Then Wait for Incoming Requests from the Clients.

This method tells the orb that the implementation object has been created and is ready to take requests from the client. The programming details for this explanation was given details in Code sample 5.11.

Code sample 5.11 Wait for Incoming Requests from the Clients..

```
// Then wait for incoming requests
        try{
                System.out.println("The server is ready...");
                boa.impl_is_ready();

        }//end of try

        .....
```

I. WRITING THE TRANSACTIONAL OBJECT (REGISTER)

There are a few tasks to complete to implement the transactional (Register) object:

- Derive the RegistrationImpl class from the __RegisterImplBase class.
- Implement the Register object with implementations for the add_course(...) and drop_course(...) methods that activates the Database Connection.

1. Understanding the RegistrationImpl Class Hierarchy

The RegistrationImpl class that was implemented is derived from the _RegisterImplBase class that was generated by the idl2java compiler.The _RegisterImplBase interface is in turn derived from the TransactionalObject interface. *Project :: Register* inherits from *CosTransactions :: TransactionalObject* so that the transaction context is propagated to it automatically by the OTS Transaction Service.

2. Implementing the RegistrationImpl Object and its Methods

As shown in Code sample 5.12, the RegistrationImpl interface defines its constructor which creates an *RegistrationImpl* object with the *lock* and *orb* parameters. The *Orb* object is created by Server in it's main. Lock is defined by another class to allow concurrent access to the methods of RegistrationImpl class. Resource object will be explained later in this chapter.

Code sample 5.12 Constructor for the Account object

```
public RegistrationImpl(org.omg.CORBA.ORB orb)
{
        this._orb = orb;
        lock = new Lock();
        if( _orb == null ) {
            System.err.println("orb is not of expected type");
         }//end of if
        r = new myResource();
        orb.connect( r );
}//end of RegistrationImpl constructor.
```

3. Implementing Methods of the RegistrationImpl Object

The RegistrationImpl object defines some useful functions that will be used by other methods for simplicity, these methods have been created for use by other programmers to make their program efficient. These methods may be implemented as java classes in future or maybe inserted into *CosTransactions.jar* file to make the programmers more efficient.

As shown in Code sample 5.13.a, the RegistrationImpl object implements a *markForTransactionBegin* () method. When invoked, this method starts a new transaction by locking this object until it is unlocked by the *markForRollback*() or *markForCommit*() method. The *markForTransactionBegin* method gets the current object from another user defined method named *getCurrent*().

A resource must be registered with the Transaction Service. Since we have our own resource implementation (*myResource.java*), we must take care of the registration of the resource with the transaction service. A *Current* object is obtained using the CORBA bootstrap mechanism. From *Current* we get the *Control* object, and from it the *Coordinator* object. The resource can then be registered with the Coordinator object.

Code sample 5.13.a Implementation of the markForTransactionBegin method

```
public void markForTransactionBegin(){
// This operation register a resource and mark transaction to be rollbacked only
        try {
        //lock add_course
                lock.lock();
                // get current
                current = getCurrent();
                System.out.println("start transaction");
                current.begin();
```

```
                    // get control and coordinator
                    String transName =   current.get_transaction_name();
                    System.out.println("RegistrationImp:name of the transaction : "  + transName);
                    org.omg.CosTransactions.Status currentStatus = current.get_status();
                    System.out.println("RegistrationImpl: status of the transaction: " +
                                        currentStatus.value());
                    System.err.println("RegistrationImpl: get control");
                    Control control = current.get_control();
                    System.err.println("RegistrationImpl: get coordinator");
                    coordinator = control.get_coordinator();
                    // register resource
                    System.out.println("RegistrationImpl: register resource with OTS");
                    recCoordinator =    coordinator.register_resource( r );
            }// end of try block
            catch ( java.lang.Exception e ){
                    System.err.println("markForTransactionBegin catch block\n" + e);
            }
    }//end of markForTransactionBegin method
```

Code sample 5.13.b shows the implementation of markForRollback() method. When invoked, this method calls rollback_only() to force the transaction originator to rollback the transaction.

Code sample 5.13.b Implementation of the markForRollback method

```
public void markForRollback(){
        try {
                //unlock add_course
                lock.unlock();
                coordinator.rollback_only();
        }//end of try block
```

```
        catch ( java.lang.Exception e ){
        System.err.println(e);
throw new org.omg.CORBA.BAD_PARAM();
}//end of catch block
}//end of markForRollback()
```

Code sample 5.13.c shows the implementation of a markForCommit () method. When invoked, this method calls commit() to force the transaction originator to commit the transaction. This method also unlocks the transactional object.

Code sample 5.13.c Implementation of the markForCommit method

```
public void markForCommit() {
        try {
                //unlock add_course
                 lock.unlock();
                org.omg.CosTransactions.Status currentStatus2 = current.get_status();
                System.out.println("RegistrationImpl: status of the transaction:"+
                currentStatus2.value());
                current.commit(true);
                System.out.println("Transaction has been Committed");
                org.omg.CosTransactions.Status currentStatus3 = current.get_status();
                System.out.println("RegistrationImpl: status of the transaction: " +
                 currentStatus3.value());
        } //end of try block
        catch ( java.lang.Exception e ) {
                System.err.println(e);
                markForRollback();
        }//end of catch block
}//end of markForCommit()
```

Code sample 5.13.d shows the implementation of a *getCurrent* () method. When invoked, this method calls getCurrent () to force the current object be created and transmitted to the caller.

Code sample 5.13.d Implementation of the getCurrent method

```
private org.omg.CosTransactions.Current getCurrent() {
        System.err.println("Session: resolve transaction current");
        try {
                org.omg.CORBA.Object obj =
                                _orb.resolve_initial_references("TransactionCurrent");
                current = org.omg.CosTransactions.CurrentHelper.narrow( obj );
                if( current == null ) {
                        System.err.println("current is not of expected type");
                }//end of if
                return current;
        }//end of try
        catch ( java.lang.Exception e ) {
                System.err.println(e);
                return null;
        }//end of catch block
}//end of getCurrent()
```

4. Implementing the Lock Object

In the Registration implementation a lock is used to serialize the access to the object. As shown in Code sample 5.14, the RegistrationImpl object also implements a *lock*() or *unlock*() method. The lock is essentially a *Boolean* variable, which indicates if an object is *locked* or not. The *lock* method sets the lock variable to true. If the object is already locked , it waits for a java event *notification*, which is caused by the *unlock*() method using *notifyAll*(). The *unlock*() method also sets the lock variable back to false.

Code sample 5.14 Implementing the Lock Object

```
package transaction.util;
public class Lock {
        private boolean locked;
        public Lock() {
                locked = false;
        }
```

```
        synchronized public void lock() {
                while(locked) {
                        try {
                                this.wait();
                        }
                        catch(InterruptedException e) {
                        }
                }
                locked = true;
        }

        synchronized public void unlock() {
                locked = false;
                        this.notifyAll();
                }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

A. CORBA OBJECT TRANSACTION SERVICE

What is an ideal programming environment for the application programmers? The answers may be various and dependent on the requirements of the programmers. However, one thing is sure, that by some software techniques, we can prevent some programmers from "*reinventing the wheel*" repeatedly in the same domain.

The concept of transactions is not only useful in database applications, but is also useful in building robust distributed mission-critical applications. Over the years, it has been shown that distributed applications can be built effectively using distributed object technology for its strong support of heterogeneous platforms. The *Object Transaction Service* (OTS) specification advocated by OMG's CORBA standard defines the fundamental transaction service to support rapid development of transactional applications in a distributed object environment. This thesis presents an implementation of the Object Transaction Service (OTS).

CORBA defines a common software interconnection bus and forms a basic building block for distributed object computing. It does really make sense if we can provide some common object services that programmers can integrate their own code with these services to develop a mission critical client/server software easily. The OTS is our first attempt. To support a complete transaction service, our work can be divided into two parts. On one hand, we provide transaction factories that will create a new transaction coordinator for each transactional client's request. The client contacts its own transaction coordinator to process the transaction. The transaction coordinator is based on a presume abort two-phase commit protocol and it will store certain information in stable storage when it makes the decision to commit or rollback the transaction. Thus, if some failures occur after the transaction has a consensus outcome, the transaction will recover later and complete the remaining work.

On the other hand, the implementations of the server objects also have to follow some principles, thus they can act properly during the transaction processing. By using inheritance, we also reduce the work to implement transactional server objects and still keep the flexibility.

There are still some ways to improve our transaction service. We only support the flat transaction model now, and the ability to handle the nested transaction model can be added to this transaction service in the future. Another aspect of the transaction service that can be improved in our next version is the propagation of the transaction context. We must pass the transaction context explicitly in each transactional function now. This is a burden for the application programmer.

The OMG also has defined the standards of other object services, e.g. the life-cycle service, the persistent service, the event service, the concurrency service and so on[2]. By specifying these services, the OMG tries to provide a way to build custom middleware. In the near future, application programmers may only have to choose the proper object services and integrate them with their own programs to build robust distributed software systems.

B. JAVA DATABASE CONNECTIVITY TRANSACTION SUPPORT

In the CORBA and Java -application model, access to database can be accomplished in much the same manner. Java, however, brings a new variant into play, Java Database Connectivity (JDBC). JDBC is the Java interface to the Standard Query Language (SQL) and is implemented in the java.sql package. Most database vendors, as well as several third-party providers, sell JDBC drivers. Drivers are either direct, sitting on top of the database's native interface, or ODBC-bridged, mapped to an ODBC implementation for a particular database.

The big advantage of using JDBC, of course, is its inherent cross-database, cross-platform capability. This makes any JDBC program theoretically portable to dozens of major SQL databases and platforms without any code rewrite.

It is possible to use an ORB implementation to talk directly through a database native client library or ODBC, given that the proper IDL definition exists.

Transaction support is an important element of large-scale, enterprise-wide deployments. In recent years, the definition of a transaction has broadened from simple database operations to include operations against a variety of network objects. In fact, any group of operations that acts together in a logical, dependent manner can be viewed as a transaction. If any one of the object services fails, the entire transaction fails, and any previous operations need to be rolled back.

CORBA and JDBC both provide transaction support. CORBA services includes a transaction service that supports single- and two-phase commit, rollback, and nested transactions. JDBC has built-in transaction support; new database statements are automatically committed after each executes successfully. Finer-grain control for multiple statements and rollback can be achieved by using the JDBC Connection object's setTransactionIsolation( ) method in conjunction with the commit( ) and rollback( ) methods. The JDBC Connection object supports five distinct transaction levels, which may be supported wholly or in part by the underlying RDBMS.

JDBC is currently limited in that it cannot manage transactions across multiple connections. For transaction support across databases or object services, CORBA's Transaction Service can provide the correct level of abstraction.

Additionally, there are several restrictions that must be enforced to ensure that the OTS Transaction Service can manage transactions.

The complete list of restrictions is as follows:

- Only one application server can be involved in a transaction.
- Only one Resource may be involved in a transaction. The JDBC DirectConnect driver transparently registers this. Resource--the application may not register any other Resources for the transaction.

- Applications must obtain a JDBC connection when required, and then call close() on that connection when that particular unit of work is completed. Since connections are pooled, calling close() on the connection is inexpensive, and does not really result in closing the underlying database connection.

- Since the JDBC connections are pooled, properties that are set for connections in a particular transaction will remain in effect when the connections are reused by other transactions.

APPENDIX A: IDL DEFINITION FOR COSTRANSACTION

```
#ifndef _COS_TRANSACTION_
#define _COS_TRANSACTION_

#pragma prefix "omg.org"

module CosTransactions {

      enum Status {
            StatusActive,
            StatusMarkedRollback,
            StatusPrepared,
            StatusCommitted,
            StatusRolledBack,
            StatusUnknown,
            StatusNoTransaction
      };

      enum Vote {
            VoteCommit,
            VoteRollback,
            VoteReadOnly
      };

      exception TransactionRequired {};
      exception TransactionRolledBack {};
      exception InvalidTranscation {};
      exception HeuristicRollback {};
      exception HeuristicCommit {};
      exception HeuristicMixed {};
      exception HeuristicHazard {};
      exception WrongTransaction {};
      exception SubtransactionsUnavailable {};
      exception NotSubtransaction {};
      exception Inactive {};
      exception NotPrepared {};
      exception NoTransaction {};
      exception InvalidControl {};
      exception Unavailable {};

      interface Control;
      interface Terminator;
      interface Coordinator;
      interface Resource;
      interface RecoveryCoordinator;
      interface SubtransactionAwareResource;
      interface TransactionFactory;
      interface TransactionObject;
      interface Current;
```

```
interface Current {
      void begin() raises(SubtransactionsUnavailable);
      void commit(in boolean report_heuristics) raises
            (NoTransaction, HeuristicMixed, HeuristicHazard);
      void rollback() raises(NoTransaction);
      Status get_status();
      string get_transaction_name();
      void set_timeout(in unsigned long seconds);
      Control get_control();
      Control suspend();
      void resume(in Control which) raises(InvalidControl);
};
interface TransactionFactory {
      Control create(in unsigned long time_out);
};
interface Control {
      Terminator get_terminator() raises (Unavailable);
      Coordinator get_coordinator() raises (Unavailable);
};
interface Terminator {
      void commit (in boolean report_heuristics)
            raises(HeuristicMixed,HeuristicHazard);
      void rollback();
};
interface Coordinator {
      Status get_status();
      Status get_parent_status();
      Status get_top_level_status();
      boolean is_same_transaction(in Coordinator tc);
      boolean is_related_transaction(in Coordinator tc);
      boolean is_ancestor_transaction(in Coordinator  tc);
      bolean is_descendant_transaction(in Coordinator tc);
      boolean is_top_level_transaction();
      unsigned long hash_transaction();
      unsigned long hash_top_level_tran();
      RecoveryCoordinator register_resource(in Resource
                                          r)raises(Inactive);
      void register_subtran_aware(in SubtransactionAwareResource
                                          r) raises(Inactive);
      void rollback_only() raises (Inactive);
      string get_transaction_name();
      Control create_subtransaction() raises
                  (SubtransactionsUnavailable, Inactive);
};
interface RecoveryCoordinator {
      Status replay_completion(in Resource r)
                        raises (NotPrepared);
};
```

```
        interface Resource {
                Vote prepare();
                void rollback() raises(HeuristicCommit,
                                HeuristicMixed,HeuristicHazard);
                void commit() raises (NotPrepared,
                        HeuristicRollback,HeuristicMixed, HeuristicHazard);
                void commit_one_phase() raises(HeuristicRollback,
                         HeuristicMixed,HeuristicHazard);
                void forget();
        };
        interface SubtransactionAwareResource : Resource {
                void commit_subtransaction(in Coordinator parent);
                void rollback_subtransaction();
        };
        interface TransactionalObject{
        };
};

module CosTSInteroperation
{
        struct otid_t
        {
                long formatID;
                long bequal_length;
                sequence<octet> tid;
        };
        struct TransIdentity
        {
                CosTransactions::Coordinator coordinator;
                CosTransactions::Terminator terminator;
                otid_t otid;
        };
        struct PropagationContext
        {
                unsigned long timeout;
                TransIdentity current;
                sequence<TransIdentity> parents;
                any implementation_specific_data;
        };
};
#endif
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: IDL DEFINITION FOR PROJECT

```
#include "CosTransactions.idl"

module Project{

   exception UnavailableCourse
     {
        string course_name;
     };
        struct course
        {
                string index_number;
                string course_code;
                string section;
                string course_name;
                string meeting;
                string credits;
                string student_ID;
        };
        typedef sequence<course>courseSeq;

        interface Register :  CosTransactions::TransactionalObject
        {
                courseSeq add_course(in string course_index_number, in
                        string Student_ID, in string password ,
                              in string studentName)
                               raises ( UnavailableCourse );
                courseSeq drop_course(in string course_index_number,
                              in string Student_ID, in string password,
                              in string studentName)
                                raises ( UnavailableCourse );
        };
};
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: SERVER IMPLEMENTATION

```
//-------------------------------------------------------------------
// Filename :     Server.java
// Date     :     01 Jan 2000
//Subject   :     Master Thesis
// Compiler :     Sun JDK1.3
//-------------------------------------------------------------------

public class Server  {

public static void main(String args[]) throws Exception  {

      if( args.length != 0) {
            System.err.println("java -DJAVA_ORB_DIR=H:/JavaORB/bin/
            Server ");
            System.exit( 1 );
      }//end of if

      System.out.println("Starting the ORB...");

      // Initialize the ORB
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
                  org.omg.CosTransactions.Boot.init( orb );

      // Initialize the BOA
      org.omg.CORBA.BOA boa = org.omg.CORBA.BOA.init(orb,args);

      // create and register the object
      // Then declare the transactional
      RegistrationImpl orblet = new RegistrationImpl(orb);

      // Connect the instance to BOA
      boa.connect(orblet);

      // Export into a file the object reference
      String reference = orb.object_to_string(orblet);
      JavaORB.util.IORManager ior_manager  =     new
      javaORB.util.IORManager();
      JavaORB.util.IORBag ior_bag = null;
      ior_bag = ior_manager.extract( reference );

      System.out.println("");
      System.out.println("IOR dump result");
      System.out.println("");
      System.out.println("IIOP version : " + ior_bag.version.major +
                        "." + ior_bag.version.minor );
      System.out.println("Host name : " + ior_bag.host );
      System.out.println("Port number : " + ior_bag.port );
      System.out.println("Object id : " + ior_bag.id );
      System.out.println("Object key : " + new String( ior_bag.key ) );
      System.out.println("");
```

```
        //then put it into a file
        try
        {
              java.io.FileOutputStream file = new
                          java.io.FileOutputStream("ObjectId");
              java.io.PrintStream pfile=new java.io.PrintStream(file);
              pfile.println(reference);
        }//end of try
        catch ( java.io.IOException ex )
        {
              System.out.println("Unable to export server reference");
        }//end of catch
        // Then wait for incoming requests
        try
        {
              System.out.println("The server is ready...");
              boa.impl_is_ready();
        }//end of try
        catch ( org.omg.CORBA.SystemException e )
        {
              System.out.println("An exception has been intercepted.");
        }//end of catch
    }//end of main
}//end of server
```

APPENDIX D: CLIENT IMPLEMENTATION

```
//--------------------------------------------------------------------
// Filename :       Client.java
// Date            :      01 Jan 2000
//Subject     :     Master Thesis
// Compiler :       Sun JDK1.3
//--------------------------------------------------------------------

import java.applet.Applet;
import java.awt.*;
import java.util.*;
import java.awt.event.*;
import java.lang.*;
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
import javax.swing.border.*;
import org.omg.CORBA.*;
import org.omg.CORBA.ORBPackage.*;
import org.omg.Project.*;
import org.omg.CosTransactions.*;

public class Client implements LayoutManager
{
    static String[] ConnectOptionNames = { "Connect" };
    static String   ConnectTitle = "Connection Information";
    Dimension   origin = new Dimension(0, 0);
    JButton     addCourseButton;
    JButton     dropCourseButton;
    JButton     showRegisteredCoursesButton;
    JPanel      connectionPanel;
    JFrame      frame;
    JLabel      userNameLabel;
    JTextField  userNameField;
    JLabel      passwordLabel;
    JTextField  passwordField;
    JComponent  queryAggregate;
    JPanel      mainPanel;
    JScrollPane tableAggregate;
    JLabel courseIndexNumberLabel1;
    JLabel courseIndexNumberLabel2 ;
    JLabel addCourseStudentNameLabel ;
    DefaultTableModel dataModel;
    JTextField addCourseTextField;
    JTextField dropCourseTextField;
    JTextField studentNameTextField;

     //Now get the Transactional server reference
    static Register orblet ;
    static String student_ID = null;
    static String password = null;
    final String[] names = {"Index Number", "Course Code", "Section",
                    "Course Name", "Meeting", "Credits","Student ID"};
```

```
static course[] data = new course[100];
static org.omg.CORBA.Object obj ;
static org.omg.CORBA.ORB orb;
org.omg.CosTransactions.Current current = null;


  public Client(){
        mainPanel = new JPanel();
        for (int i=0;i<data.length-1;i++){
              data[i]= new course();
              data[i].index_number = "";
              data[i].course_code = "";
              data[i].section = "";
              data[i].course_name = "";
              data[i].meeting = "";
              data[i].credits = "";
              data[i].student_ID = "";
        }// end of for
        // Create the panel for the connection information
        createConnectionDialog();
        // Create the buttons.
        showRegisteredCoursesButton = new JButton("Show Schedule");
        addCourseButton = new JButton("Add Course");
        dropCourseButton = new JButton("Drop Course");
        //Create the labels.
        courseIndexNumberLabel1 = new JLabel("Course Index
                                              Number");
        courseIndexNumberLabel2 = new JLabel("Course Index
                                              Number");
        addCourseStudentNameLabel =    new JLabel("Student Name");
        addCourseTextField = new JTextField(6);
        dropCourseTextField = new JTextField(6);
        studentNameTextField = new  JTextField(16);
        showRegisteredCoursesButton.addActionListener(new
                                              QueryChangeListener());
        addCourseButton.addActionListener(new
                                              QueryChangeListener());
        dropCourseButton.addActionListener(new
                                              QueryChangeListener());
        // Create the table.
        tableAggregate = createTable();
        tableAggregate.setBorder(new
                                BevelBorder(BevelBorder.LOWERED));
        // Add all the components to the main panel.
        mainPanel.add(addCourseButton);
        mainPanel.add(dropCourseButton);
        mainPanel.add(showRegisteredCoursesButton);
        mainPanel.add(tableAggregate);
        mainPanel.add(addCourseTextField);
        mainPanel.add(dropCourseTextField);
        mainPanel.add(studentNameTextField);
        mainPanel.add(courseIndexNumberLabel1);
        mainPanel.add(courseIndexNumberLabel2);
        mainPanel.add(addCourseStudentNameLabel);
        mainPanel.setLayout(this);
        // Create a Frame and put the main panel in it.
```

```
        frame = new JFrame("Register a Course");

        frame.addWindowListener(new WindowAdapter() {

              public void windowClosing(WindowEvent e)
              {
                    System.exit(0);}
              }
        );
        frame.setBackground(Color.lightGray);
        frame.getContentPane().add(mainPanel);
        frame.pack();
        frame.setVisible(false);
        frame.setBounds(200, 200, 640, 480);
        activateConnectionDialog();
}//end of client class

/**
  * Brigs up a JDialog using JOptionPane containing the
  *  connectionPanel. If the user clicks on the
  * 'Connect' button the connection is reset.
  */

void activateConnectionDialog(){
      if(JOptionPane.showOptionDialog(tableAggregate,
            connectionPanel,ConnectTitle,
            JOptionPane.DEFAULT_OPTION,
            JOptionPane.INFORMATION_MESSAGE,null,
            ConnectOptionNames, ConnectOptionNames[0]) == 0)
      {
            connect();
            frame.setVisible(true);
      }
      else if(!frame.isVisible())
            System.exit(0);
      }

  /**
  * Creates the connectionPanel, which will contain all the
  * fields for the connection information.
  */
public void createConnectionDialog() {
   // Create the labels and text fields.
   userNameLabel = new JLabel("User name: ", JLabel.RIGHT);
   userNameField = new JTextField("");
   passwordLabel = new JLabel("Password: ", JLabel.RIGHT);
   passwordField = new JPasswordField("");
   connectionPanel = new JPanel(false);
   connectionPanel.setLayout(new BoxLayout(connectionPanel,
                                    BoxLayout.X_AXIS));
   JPanel namePanel = new JPanel(false);
   namePanel.setLayout(new GridLayout(0, 1));
   namePanel.add(userNameLabel);
   namePanel.add(passwordLabel);

   JPanel fieldPanel = new JPanel(false);
```

```
   fieldPanel.setLayout(new GridLayout(0, 1));
   fieldPanel.add(userNameField);
   fieldPanel.add(passwordField);

   connectionPanel.add(namePanel);
   connectionPanel.add(fieldPanel);
}//end of createConnectionDialog method

class ShowConnectionInfoListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      activateConnectionDialog();
   }//end of actionPerformed  method
}//end of ShowConnectionInfoListener method

class QueryChangeListener implements ActionListener {
   public void actionPerformed(ActionEvent e)
    {
      String c = e.getActionCommand();
      if (c.equals("Add Course")) {
            addCourse();
      }//end of if
      else if (c.equals("Drop Course")){
            dropCourse();
      }//end of else if
      else if (c.equals("Show Schedule")) {
            displaySchedule();
      }//end of else if
    }//end of actionPerformed  method
}//end of QueryChangeListener  class

public void connect(){
      student_ID = userNameField.getText();
      password = passwordField.getText();
}//end of connect  method

public void addCourse(){
      try {
            // get current
            org.omg.CosTransactions.Current current =
                  getCurrent();
            // Begin a transaction
            System.out.println("Begin a transaction.");
            current.begin();
            System.out.println(addCourseTextField.getText());
            data = orblet.add_course
                  (addCourseTextField.getText(),student_ID,
                                                      password,
            studentNameTextField.getText());
            System.out.println("!!!!" + data[0].student_ID );
            System.out.println("Transaction Name" +
            current.get_transaction_name());
            System.out.println(" Commit the transaction ");
            current.commit(true);
            System.out.println("The transaction has
                              been commited");
      }//end of try
```

```
        catch ( java.lang.Exception e ){
              System.err.println("No transaction - rollback:\n " +
                                                              e );

        try {
              System.out.println("Current Transaction is rolling
                                                          back");
              current.rollback();
        }//end of try
        catch( org.omg.CosTransactions.NoTransaction nt) {
              System.err.println("\nNo transaction "   + nt );
              System.exit( 1 );
        }//end of second catch
  }//end of first catch
}//end of add course method
 public void dropCourse() {
        try {
        // get current
              org.omg.CosTransactions.Current current =
                                                  getCurrent();
              System.out.println("Begin a transaction.");
              current.begin();
              System.out.println(dropCourseTextField.getText());
              data =
                    orblet.drop_course
                    (dropCourseTextField.getText(),student_ID,
                    password, studentNameTextField.getText());
              System.out.println("!!!!!!" + data[0].index_number );
              System.out.println("In dropCourse");
              System.out.println("Call Commit the transaction");
              System.out.println("Transaction Name" +
              current.get_transaction_name());
              // Commit the transaction
              current.commit(true);
              System.out.println("The transaction has been
                                commited");
        }//end of try
        catch ( java.lang.Exception e ){
              System.err.println("No transaction-rollback:\n "+e );
              try {
                    System.out.println("Current Transaction is
                                rolling back");
                    current.rollback();
              }//end of try

              catch( org.omg.CosTransactions.NoTransaction nt) {
                    System.err.println("\nNo transaction "  + nt );
                    System.exit( 1 );
              }//end of second catch
        }//end of first catch
 }//end of dropCourse method

 private org.omg.CosTransactions.Current getCurrent() {
        System.err.println("Session: Resolve TransactionCurrent to
                          get access to Current object.");
        try {
```

```
            org.omg.CORBA.Object obj =
            orb.resolve_initial_references("TransactionCurrent");
            org.omg.CosTransactions.Current current =
            org.omg.CosTransactions.CurrentHelper.narrow( obj );

            if( current == null ) {
                  System.err.println("current is not of expected
                                                      type");
            }//end of if
            return current;
      }//end of try
      catch ( java.lang.Exception e )
      {
            System.err.println(e);
            return null;
      }//end of catch block
}//end of getCurrent()

 public void displaySchedule() {
      System.out.println("Display Schedule" );
      dataModel.fireTableDataChanged();
      //createTable();
}//end of displaySchedule method
 public JScrollPane createTable() {
      System.out.println("createTable method is called");
      // Create a model of the data.
      // TableModel dataModel = new AbstractTableModel() {
      dataModel = new DefaultTableModel() {
            // These methods always need to be implemented.
            public int getColumnCount() { return names.length; }
            public int getRowCount() { return data.length;}
            public java.lang.Object getValueAt(int row, int col)
            {

                  if (data[0].index_number.equals(""))
                        return "";
                  if(col==0)
                        return data[row].index_number;
                  else if(col==1)
                        return data[row].course_code;
                  else if(col==2)
                        return data[row].section;
                  else if(col==3)
                        return data[row].course_name;
                  else if(col==4)
                        return data[row].meeting;
                  else if(col==5)
                        return data[row].credits;
                  else if(col==6)
                        return data[row].student_ID;
                  return "";
            }//end of getValueAt method
            // The default implementations of these methods in
            // AbstractTableModel would work, but we can refine
            //them.
            public String getColumnName(int column)
```

```
            {
                  return names[column];
            }


            public boolean isCellEditable(int row, int col)
            {
                  return (col==4);
            }
            public void setValueAt(java.lang.Object aValue,
                                    int row, int col)
            {
                  if(col==1)
                        data[row].index_number = (String)aValue;
                  else if(col==2)
                        data[row].course_code = (String)aValue;
                  else if(col==3)
                        data[row].section = (String)aValue;
                  else if(col==4)
                        data[row].course_name = (String)aValue;
                  else if(col==5)
                        data[row].meeting = (String)aValue;
                  else if(col==6)
                        data[row].credits = (String)aValue;
                  else if(col==7)
                        data[row].student_ID = (String)aValue;
            }//end of setValueAt method
      }; //end of DefaultTableModel  definition

      JTable tableView = new JTable(dataModel);
      JScrollPane scrollpane =
                  JTable.createScrollPaneForTable(tableView);
      return scrollpane;
}//end of JScrollPane createTable method

public static void main(String args[]) {
      JavaORB.Trace.setTraceFile("bug.log");
      JavaORB.Trace.setTraceLevel(4);
            try  {

                  // 1.
                  // Initialize the ORB
                  orb = org.omg.CORBA.ORB.init(args,null);
                  org.omg.CosTransactions.Boot.init( orb );
                  // 2.
                  // Get the Calculator reference from a file
                  orblet = null;
                  obj = null;
                  try{
                        java.io.FileInputStream file = new
                        java.io.FileInputStream("ObjectId");
                        java.io.DataInputStream myInput = new
                        java.io.DataInputStream(file);
                        String stringTarget = myInput.readLine();
```

```
                        obj = orb.string_to_object(stringTarget);

                        JavaORB.util.IORManager ior_manager =
                              new JavaORB.util.IORManager();
                        JavaORB.util.IORBag ior_bag = null;
                        ior_bag = ior_manager.extract( obj );
                        System.out.println("");
                              System.out.println("IOR dump
                                                      result");
                              System.out.println("");
                              System.out.println("IIOP version :
                               +    ior_bag.version.major + "." +
                                    ior_bag.version.minor );
                              System.out.println("Host name : " +
                                    ior_bag.host );
                              System.out.println("Port number : "
                                    +     ior_bag.port );
                              System.out.println("Object id : " +
                                    ior_bag.id );
                              System.out.println("Object key : "
                                    + new String( ior_bag.key ));
                              System.out.println("");
                              JavaORB.Trace.setTraceFile
                                                ("bug.log");
                              JavaORB.Trace.setTraceLevel(4);
                  }//end of try
                  catch (java.io.IOException ex )
                  {
                        System.out.println("File error");
                        System.exit(0);
                  }//end of catch

                  // 3.
                  // Narrow the object reference
                  orblet = RegisterHelper.narrow(obj);
                  // Use the client object
                  new Client();
            }//end of try
            catch (Exception e)
            {
                  System.out.println("Cannot connect to ORB for
                                                      Register");
                  return;
            }//end of catch
}//end of main

public Dimension preferredLayoutSize(java.awt.Container c){
            return origin;
}
public Dimension minimumLayoutSize(java.awt.Container c){
            return origin;
}
public void addLayoutComponent(String s, Component c) {}
public void removeLayoutComponent(Component c) {}
public void layoutContainer(java.awt.Container c)
{
```

```
            Rectangle b = c.getBounds();
            int topHeight = 90;
            int inset = 4;
            dropCourseButton.setBounds(b.width-2*inset-130, inset, 120,
                                                                    25);
            addCourseButton.setBounds(b.width-2*inset-130, 32, 120,
                                                                    25);
            showRegisteredCoursesButton.setBounds(b.width-2*inset-130,
                                                    60,120,25);
            dropCourseTextField.setBounds(b.width-225, inset, 60, 25);
            addCourseTextField.setBounds(b.width-225, 32, 60, 25);
            studentNameTextField.setBounds(b.width-520, 32, 125, 25);
            courseIndexNumberLabel1.setBounds(b.width-375, inset, 140,
                                                    25);
            courseIndexNumberLabel2.setBounds(b.width-375, 32, 140,
                                                    25);
            addCourseStudentNameLabel.setBounds(b.width-625, 32, 100,
                                                            25);
            tableAggregate.setBounds(new Rectangle(inset,
                                inset + topHeight,b.width-2*inset,
                                b.height-2*inset - topHeight));
      }//end of  layoutContainer  method
}//end of client class.
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E: REGISTRATION CLASS IMPLEMENTATION

```
//-------------------------------------------------------------------
//Filename :      :RegistrationImpl.java
//Date            :01 Jan 2000
//Subject         :Master Thesis
//Compiler        :Sun JDK1.3
//-------------------------------------------------------------------
import java.sql.*;
import java.util.*;
import java.net.*;
import org.omg.CORBA.*;
import org.omg.Project.*;
import org.omg.CosTransactions.*;
import transaction.util.*;

public  class RegistrationImpl extends _RegisterImplBase
{
      Connection        connection;
      Statement         statement ;
      ResultSet         resultSet ;
      String[]          columnNames = {};
      Class[]           columnTpyes = {};
      Vector            rows = new Vector();
      String      url = "jdbc:odbc:Sybase";
      String      driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
      String      databaseUser="yhazir";
      String      databasePassword="yh7093";
      ResultSetMetaData metaData;
      String[][]  classes;
      private Lock lock;

      /**
       * Reference to the ORB
       */
      org.omg.CORBA.ORB _orb ;
```

```
/**
 * Reference to the resource
 */
myResource r = null;
org.omg.CosTransactions.Current current = null;
org.omg.CosTransactions.Coordinator coordinator = null;
org.omg.CosTransactions.RecoveryCoordinator recCoordinator =
                                                        null;
public RegistrationImpl(org.omg.CORBA.ORB orb)
{
      _orb = orb;
      lock = new Lock();
      if( _orb == null ) {
            System.err.println("orb is not of expected type");
      }//end of if
      r = new myResource();
      orb.connect( r );
}//end of RegistrationImpl constructor.
/**
* add_course Operation
*
* This operation can be used in a transactional mode
*/
public synchronized course[] add_course(String
      Index_Number, String Student_ID, String
      password, String student_Name)
       throws  UnavailableCourse
      {
            boolean commit = false ;
            course[] mycourseSeq = new course[100];
            try {
                  markForTransactionBegin();
                  Class.forName(driverName);
                  System.out.println("Opening db connection in
                                          add_course");
```

```
            connection = DriverManager.getConnection(url,
                                        Student_ID, password);
            statement = connection.createStatement();
for (int i=0;i< 100;i++)
      mycourseSeq[i] = new course("", "", "",
                                        "", "", "","");
      if (connection == null || statement == null)
      {
            System.err.println("There is no database to
                        execute the query.");
            markForRollback();
            return null;
      }//end of if
      System.err.println(Index_Number);
      System.err.println(student_Name);
      String str = new String("INSERT INTO
      REGISTERED_COURSES
      (Index_Number,Course_Code,Section,
      Course_Name,Meeting,Credits )"+
      "\nSELECT   Index_Number,Course_Code,Section,
      Course_Name,Meeting,Credits " +     "\nFROM courses
      WHERE Index_Number =" + "\'" +      Index_Number +
                  "\'");
      System.out.println(str);
      int returnVal = statement.executeUpdate(str);
      str = new String("\nupdate
      REGISTERED_COURSES set  Student_ID ="
       + "\'" + student_Name + "\'" +"\nwhere
      Index_Number =" + "\'" +
      Index_Number + "\'");
      System.out.println(str);
      returnVal = statement.executeUpdate(str);
      if(returnVal == 0){
            throw new UnavailableCourse("ERROR
            IN SQL Statement or     Primary
            Key", Index_Number);
      }//end of if
```

```
resultSet = statement.executeQuery
("SELECT  Index_Number,Course_Code,
Section, Course_Name, Meeting, Credits,
Student_ID  FROM REGISTERED_COURSES ");
metaData = resultSet.getMetaData();
// Get all rows.
rows = new Vector();
int rowcount=0;
while (resultSet.next()){
      Vector newRow = new Vector();
      for (int i = 1;i <= metaData.getColumnCount();
                  i++)
      {
            if(i==1){
                  mycourseSeq[rowcount].index_number
                        = resultSet.getString(i);
                  continue;
            }//end of if
            else if(i==2){
                  mycourseSeq[rowcount].course_code =
                              resultSet.getString(i);
                  continue;
            }//end of else if
            else if(i==3){
                  mycourseSeq[rowcount].section =
                              resultSet.getString(i);
                  continue;
            }//end of else if
            else if(i==4){
                  mycourseSeq[rowcount].course_name =
                              resultSet.getString(i);
                  continue;
            }//end of else if
            else if(i==5){
                  mycourseSeq[rowcount].meeting =
                              resultSet.getString(i);
                  continue;
```

```
                    }//end of else if



                    else if(i==6){
                          mycourseSeq[rowcount].credits =
                                      resultSet.getString(i);
                          continue;
                    }//end of else if
                    else if(i==7){
                          mycourseSeq[rowcount].student_ID =
                                      resultSet.getString(i);
                          continue;
                    }//end of else if
              }//end of for
              rowcount++;
        }//end of while
        resultSet.close();
        statement.close();
        commit = true;
}//end of try
catch (UnavailableCourse uac)
{
        System.out.println("\n ERROR IN SQL Statement or
                                                      PK");
        System.err.println("\n This course is not available
                                          in database.");
        System.err.println("\n Unavailable Course Name :" +
                                      uac.course_name );
        markForRollback();
}//end of catch block
catch (SQLException sql)
{
        System.err.println("Cannot connect to this
                                          database.");
        System.err.println(sql);
```

```
            markForRollback();
      }//end of catch block
      catch (ClassNotFoundException ex) {
            System.err.println("Cannot find the database driver
                                                      classes.");
            System.err.println(ex);
            markForRollback();
      } //end of catch block
      finally {
            // Commit or rollback the transaction.
            if (commit) {
                  System.out.println("*** Committing transaction
                                                      ****");
                  markForCommit();
            }//end of if
            else
            {
                  System.out.println("*** Rolling back
                                          transaction ***");
                  markForRollback();
            } //end of else if
      }//end of finally block
      System.out.println("######################");
      for(int ix = 0; ix < 10; ix++) {
            System.out.println("Student ID=" +
                              mycourseSeq[ix].student_ID);
      }
      return new courseSeqHolder(mycourseSeq).value;
      //return  mycourseSeq;
}//end of  add_course  method
public synchronized course[] drop_course(String
                  course_index_number, String   Student_ID,
                  String password, String student_Name)
                        throws  UnavailableCourse  {
      boolean commit = false ;
      course[] mycourseSeq = new course[100];
      try {
```

```
markForTransactionBegin();
Class.forName(driverName);
System.out.println("Opening db connection in
      drop_course");
connection = DriverManager.getConnection(url,
                  Student_ID, password);
statement = connection.createStatement();
for (int i=0;i< 100;i++)
mycourseSeq[i] = new course("", "", "", "", "",
                                    "","");
if (connection == null || statement == null) {
      System.err.println("There is no database to
                  execute the query.");
      return null;
}//end of if

int returnVal = statement.executeUpdate("DELETE FROM
                        REGISTERED_COURSES
                        WHERE INDEX_NUMBER = "
                        +     "\'" +
                        course_index_number
                        +"\'" + "
                        AND Student_ID =" +
                        "\'" +
                        student_Name + "\' ;");
if(returnVal == 0){
      throw new UnavailableCourse("ERROR IN SQL
                  Statement or Primary Key",
                  course_index_number);
}//end of if
resultSet = statement.executeQuery("SELECT
                  Index_Number, Course_Code, Section,
                  Course_Name, Meeting,
                  Credits,Student_ID
                  FROM REGISTERED_COURSES ");
metaData = resultSet.getMetaData();
// Get all rows.
```

```
rows = new Vector();
int rowcount=0;
while (resultSet.next()){
      Vector newRow = new Vector();
      for (int i = 1; i <= metaData.getColumnCount();
                                                  i++)
            {

      if(i==1){
            mycourseSeq[rowcount].index_number =
                              resultSet.getString(i);
            continue;
      }//end of if
      else if(i==2){
            mycourseSeq[rowcount].course_code =
                        resultSet.getString(i);
            continue;
      }//end of else if block
      else if(i==3){
            mycourseSeq[rowcount].section =
                        resultSet.getString(i);
            continue;
      }//end of else if block
      else if(i==4){
            mycourseSeq[rowcount].course_name =
                              resultSet.getString(i);
            continue;
      }//end of else if block
      else if(i==5){
            mycourseSeq[rowcount].meeting =
                              resultSet.getString(i);
            continue;
      }//end of else if block
      else if(i==6){
            mycourseSeq[rowcount].credits =
                        resultSet.getString(i);
            continue;
```

```
                    }//end of else if block
                    else if(i==7){
                          mycourseSeq[rowcount].student_ID =
                                resultSet.getString(i);
                          continue;
                    }//end of else if block
                    System.out.println(classes[rowcount][i]);
              }//end of for
              rowcount++;
        }//end of while
        commit = true;
}//end of try
catch (UnavailableCourse uac){
        System.out.println("\n ERROR IN SQL Statement or PK");
        System.err.println("\n This course is not available in
                                database.");
        System.err.println("\n Unavailable Course Name :" +
                                uac.course_name );
        markForRollback();
}//end of catch block
catch (SQLException sql){
        System.err.println("Cannot connect to this database.");
        System.err.println(sql);
        markForRollback();
}//end of catch block
catch (ClassNotFoundException ex) {
        System.err.println("Cannot find the database driver
                          classes.");
        System.err.println(ex);
        markForRollback();
}//end of catch block
finally {
        // Commit or rollback the transaction.
        if (commit) {
              System.out.println("*** Committing transaction ***");
              markForCommit();
        }//end of if
```

```
            else{
                  System.out.println("*** Rolling back transaction
                                                      ***");
                  markForRollback();
            }//end of else if block
      }//end of finally block
      return new courseSeqHolder(mycourseSeq).value;
} //end of drop_course method


      public void markForTransactionBegin(){

            // This operation register a resource and mark transaction
                                    to be rollbacked only

            try {
                  //lock add_course
                  lock.lock();
                  // get current
                  current = getCurrent();
                  System.out.println("start transaction");
                  current.begin();
                  // get control and coordinator
                  String transName =   current.get_transaction_name();
                  System.out.println("RegistrationImp:name of the
                        transaction : "   + transName);
                  org.omg.CosTransactions.Status currentStatus =
                  current.get_status();
                  System.out.println("RegistrationImpl: status of the
                  transaction: " +  currentStatus.value());
                  System.err.println("RegistrationImpl: get control");
                  Control control = current.get_control();
                  System.err.println("RegistrationImpl: get
                                                coordinator");
                  coordinator = control.get_coordinator();

                  // register resource
```

```
            System.out.println("RegistrationImpl: register
                                    resource with OTS");
            recCoordinator =  coordinator.register_resource( r );
      }// end of try block

      catch ( java.lang.Exception e )
      {
            System.err.println("markForTransactionBegin catch
                  block\n" + e);
      }
}//end of markForTransactionBegin method
public void markForRollback(){
      try {
            //unlock add_course
            lock.unlock();
            coordinator.rollback_only();
      }//end of try block
      catch ( java.lang.Exception e )
      {
            System.err.println(e);
            throw new org.omg.CORBA.BAD_PARAM();
      }//end of catch block
}//end of markForRollback()

public void markForCommit() {

      try  {
            //unlock add_course
            lock.unlock();
            org.omg.CosTransactions.Status currentStatus2 =
                                    current.get_status();
            System.out.println("RegistrationImpl: status of the
                        transaction: " + currentStatus2.value());
            current.commit(true);
            System.out.println("Transaction has been Committed");
```

```
                org.omg.CosTransactions.Status currentStatus3 =
                                                current.get_status();
                System.out.println("RegistrationImpl: status of the
                      transaction: " +  currentStatus3.value());
          } //end of try block

          catch ( java.lang.Exception e ) {
                System.err.println(e);
                markForRollback();
          }//end of catch block
    }//end of markForCommit()


    private org.omg.CosTransactions.Current getCurrent() {

          System.err.println("Session: resolve transaction current");
          try {
                org.omg.CORBA.Object obj =
                _orb.resolve_initial_references
                                  ("TransactionCurrent");
                current =
                org.omg.CosTransactions.CurrentHelper.narrow( obj );
                if( current == null ) {
                      System.err.println("current is not of expected
                      type");
                }//end of if
                return current;
          }//end of try
          catch ( java.lang.Exception e )
          {
                System.err.println(e);
                return null;
          }//end of catch block
    }//end of getCurrent()
} //end of RegistrationImpl class
```

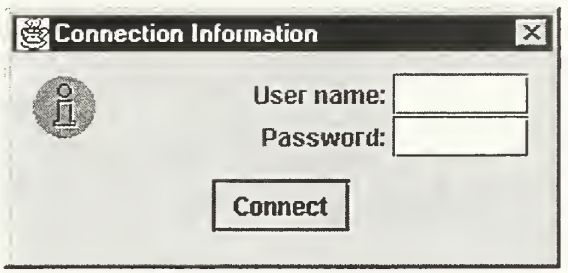
APPENDIX F: LOCK CLASS IMPLEMENTATION

```
//-------------------------------------------------------------------------
// Filename : Lock.java
// Date     : 01 Jan 2000
//Subject   : Master Thesis
// Compiler : Sun JDK1.3
//-------------------------------------------------------------------------


package transaction.util;

      public class Lock {
            private boolean locked;

            public Lock() {
            locked = false;
      }//end of  Lock constructor

      synchronized public void lock() {
            while(locked) {
                  try {
                        this.wait();
                  }//end of try
                  catch(InterruptedException e) {
                  }//end of catch
            }//end of while
            locked = true;
      }//end of lock method



      synchronized public void unlock() {
            locked = false;
            this.notifyAll();
      }//end of unlock method
}//end of Lock class.
```

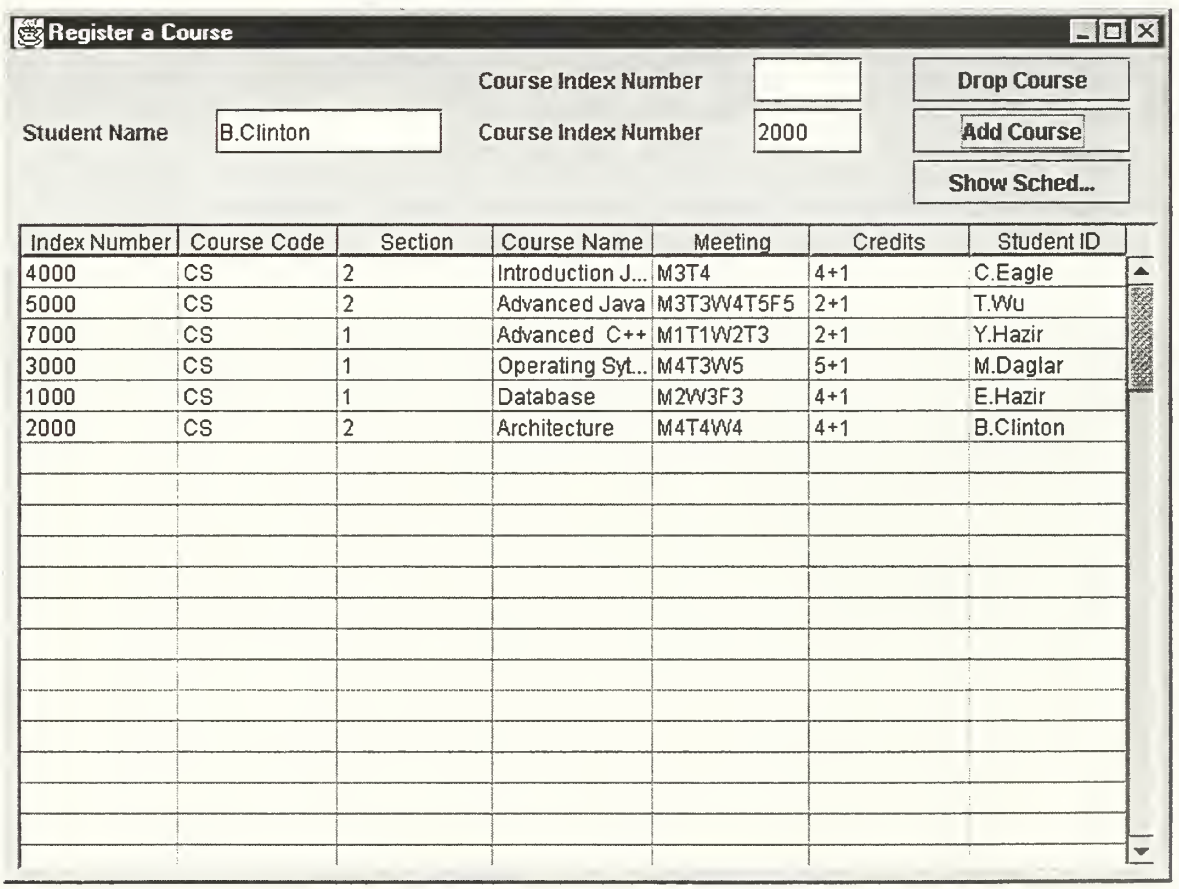
THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G: APPLICATION PROGRAM GUI SCREEN SHOTS

1. Connection To a Database

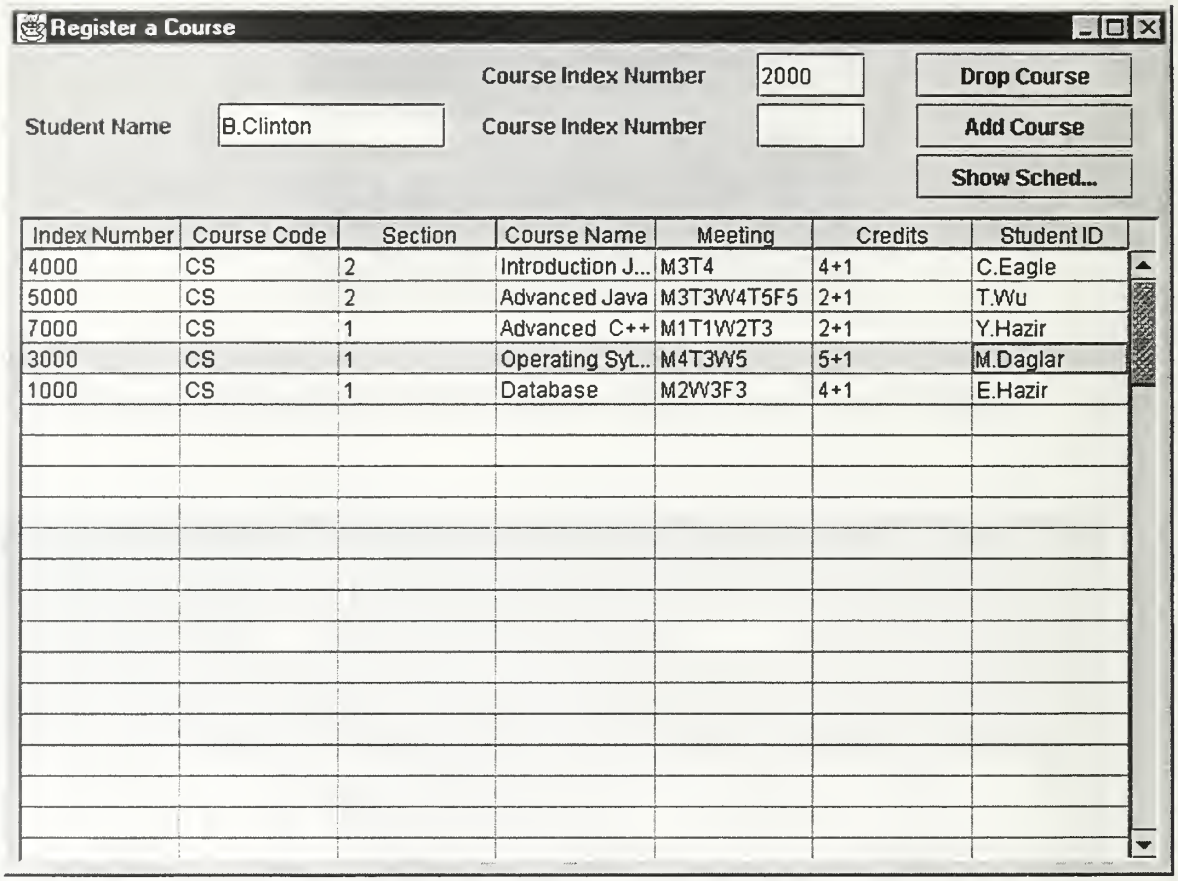


2. Adding a new course from the Course Table to Registered Course Table



Index Number	Course Code	Section	Course Name	Meeting	Credits	Student ID
4000	CS	2	Introduction J...	M3T4	4+1	C.Eagle
5000	CS	2	Advanced Java	M3T3W4T5F5	2+1	T.Wu
7000	CS	1	Advanced C++	M1T1W2T3	2+1	Y.Hazir
3000	CS	1	Operating Syt...	M4T3W5	5+1	M.Daglar
1000	CS	1	Database	M2W3F3	4+1	E.Hazir
2000	CS	2	Architecture	M4T4W4	4+1	B.Clinton

3. Dropping a course from the Registered Course Table



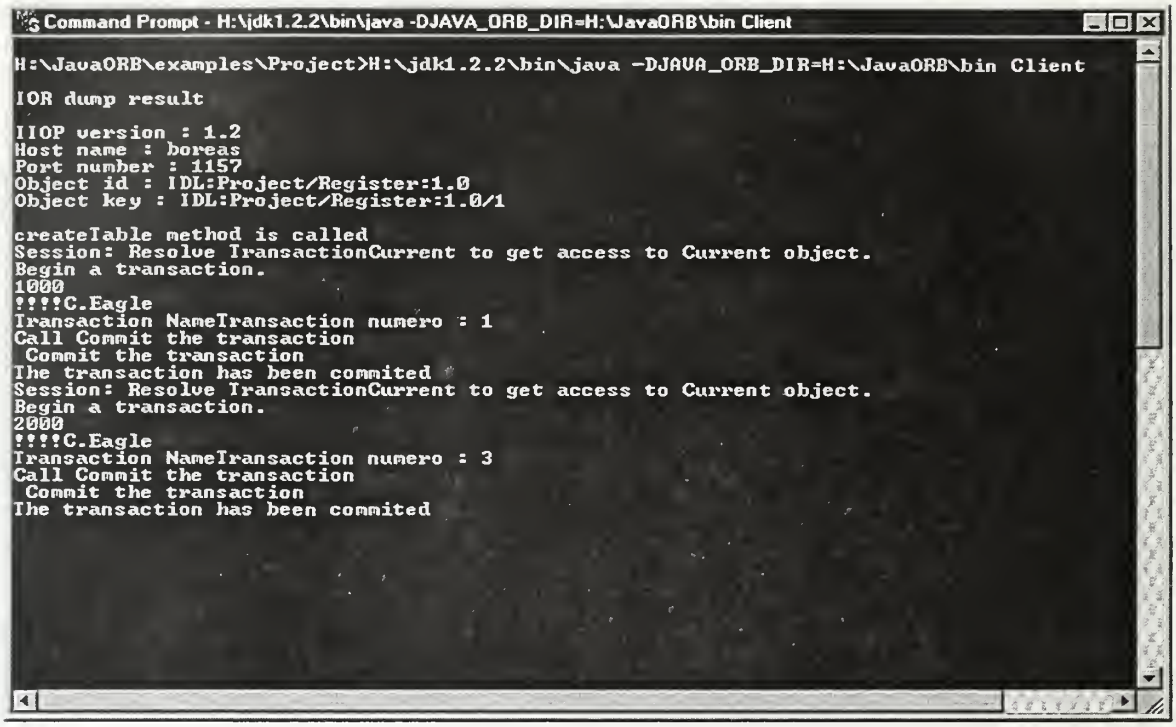
Index Number	Course Code	Section	Course Name	Meeting	Credits	Student ID
4000	CS	2	Introduction J...	M3T4	4+1	C.Eagle
5000	CS	2	Advanced Java	M3T3W4T5F5	2+1	T.Wu
7000	CS	1	Advanced C++	M1T1W2T3	2+1	Y.Hazir
3000	CS	1	Operating Syt...	M4T3W5	5+1	M.Daglar
1000	CS	1	Database	M2W3F3	4+1	E.Hazir

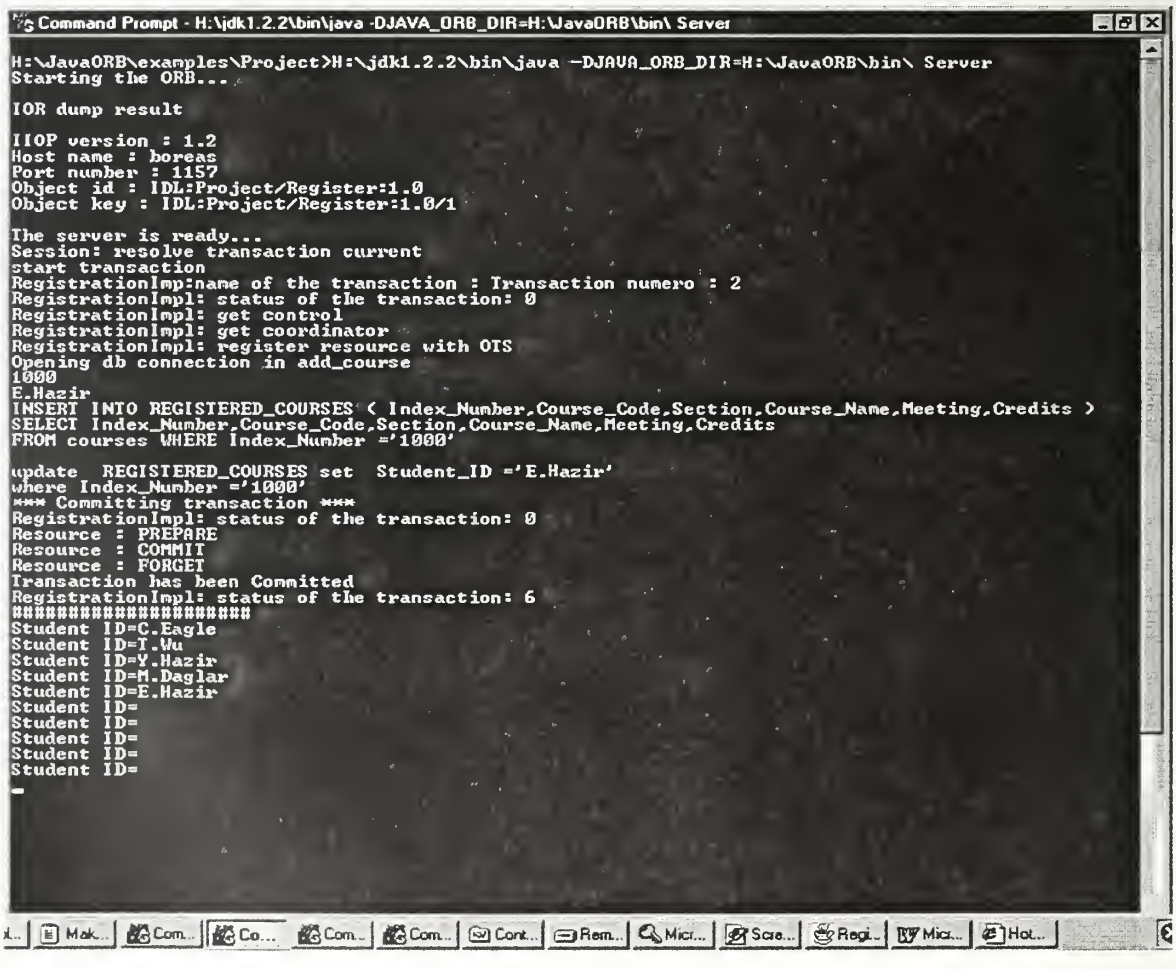
APPENDIX H: CLIENT \ SERVER DOS OUTPUT SCREEN SHOTS

1. Begin the JAVAORB Object Transaction Service(OTS)

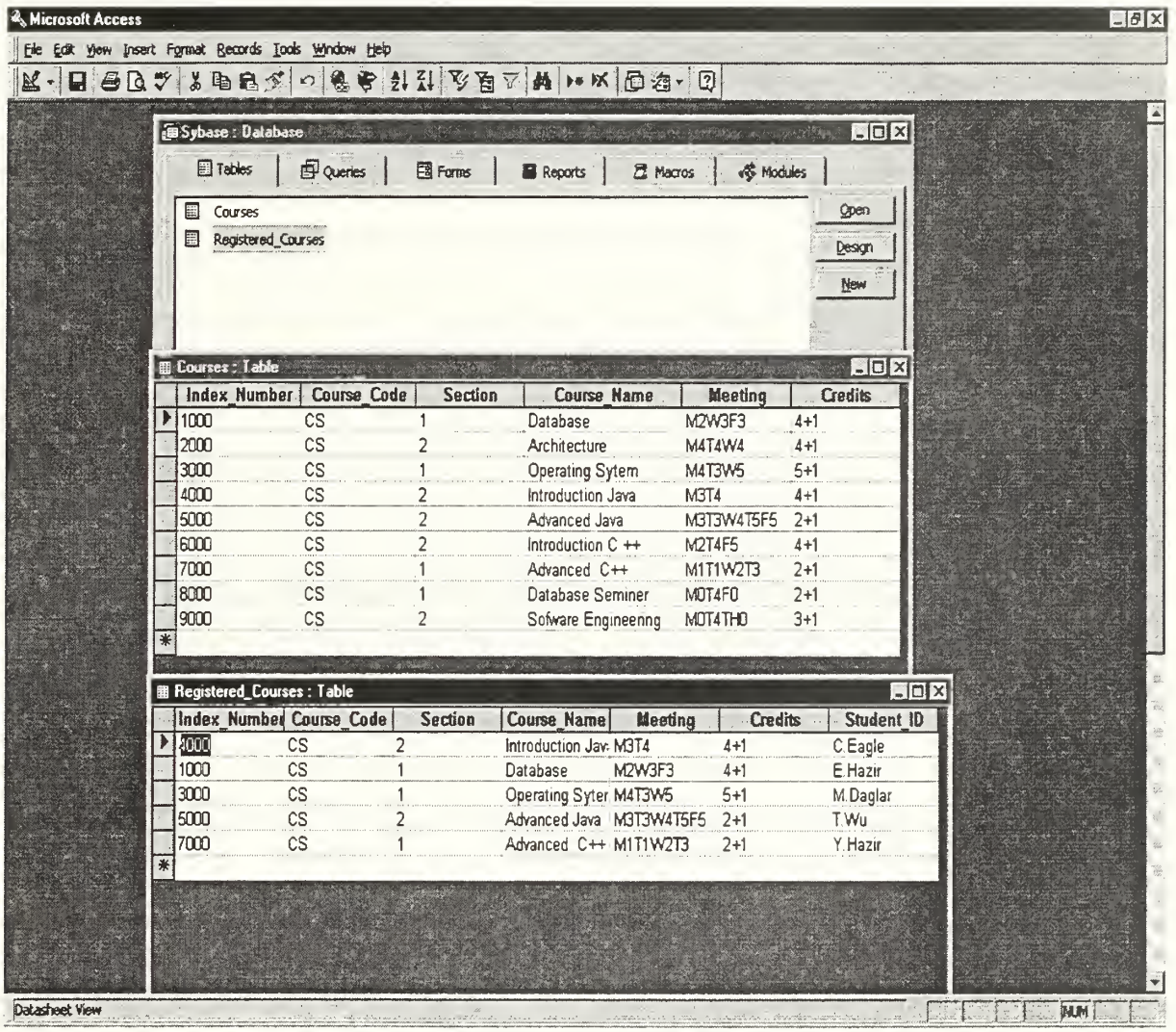
2. Start the Server Program

3. Start Client, make connection to Database and Use Add and Drop Functions





APPENDIX I: DATABASE TABLES SCREEN SHOTS



THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] "Object Management Architecture Guide ", Object Management Group, Inc., OMG TC Document 92.11.1,Revision 2.0, Sepember 1, 1992.

[2] R. Orfali and D. Harkey, "Client/Server Programming JAVA and CORBA (Second Edition).

[3] "Common Object Request Broker Architecture and Specification", Object Management Group,Inc.,Reverision 2.0,July 1995.

[4] "Components Everywhere." T. R. Halfhill and S. Salamore, BYTE, vol. 21, No. 1, pp.97-104, January 1996.

[5] "Intergration, Not Perspiration", David S. Linthicum, BYTE, vol. 21, No. 1, pp.83-96, January 1996.

[5] "Common Object Services Specification",Object Management Group,Inc.,

[6] "Distributed Systems Concepts and Design",George Coulouris, Jean Dollimore, Tim Kindberg, Addison-Wesley Publishing Company 1994.

[7] "Transaction Processing Concepts and Techniques",Jim Gray,Andeas Reuter, 1993 Morgan Kaufmann Publishers, Inc.,

[8] "Object and Transactions: together at Last",E. Cobb, OBJECT Magazine,pp.59-63, January 1995.

[9] "Client/Server with Distributed Objects",R. Orfali and D. Harkey, BYTE Magazine, pp.151-162, April 1995.

[10] "Building a Transaction Processing System on Unix Systems", S. Andrade, M.T. Carges, UniForum Conference Proceedings, February 1989.

[11] "Enterprise Transaction Processing", A. Dwyer, Uniforum Conference Proceedings, January 1991.

[12] "Distributed Computing Monitor", A. D. Wolfe, vol. 7, No. 11, November 1992.

[13] "Principles of Transaction-Oriented Database Recovery", T. Harder and A. Reuters, Computing Surveys, vol. 15, 4, 1983.

[14] "Programming guide: Orbix 2 distributed object technology",IONA Technologies Ltd. Release 2.0, November 1995.

[15] "Reference Guide: Orbix 2 distributed object technology", IONA Technologies Ltd. Release 2.0, November 1995.

[16] "IonaSphere Issue 11", IONA technologies Ltd., May 1995.

[17] "NEO Programming Guide", Beta Version, SunSoft, Inc., May 1995.

[18] "Digital's ObjetBroker-advanced Integration of Distributed Resources", Aberdeen Group Profile, November 1995.

[19] "A Dynamic and Integrated Concurrency Control for Distributed Databases", F. Pons and J. F. Vilarem, IEEE journal on Selected Areas in Communications, vol. 7, No. 3, pp. 364-373, April 1989.

[20] "Subcontract: A Flexible Base for Distributed Programming", Hamilton, M. L. Powell and J. G. Mitchell, Operating System Review, pp. 69-79, December 1993.

[21] "Transactions: Who needs Them?", Iain Houston, "http://204.146.47.71/objects/owsf.html", IBM Corporation, 1995.

[22] "TPBroker : Next-Generation Object-Oriented Transaction Processing", Hitachi, "http://www.hitachi.co.jp/Prod/comp/soft1/open-e/tpbroker/tp-head.htm"

[23] "NEO Frequently Asked Questions", SunSoft Co. "http://www.sun.com/sunsoft/neo/external/qna.html"

[24] "An Object Transaction Service Based on CORBA Architecture", Yue-Shan Chang, Yu-Ming Kao, Shyan-Ming Yuan, and Deron Liang, Proc. of 1996 IFIP/IEEE Int'l Conf. on Distributed Platforms, (Dresden, Germany), Feb. 1996,

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Deniz Kuvvetleri Komutanligi 1
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

4. Deniz Kuvvetleri Komutanligi 1
   Kutuphanesi
   Bakanliklar
   Ankara, TURKEY

5. Deniz Harp Okulu 2
   Kutuphanesi
   Tuzla
   Istanbul, TURKEY

6. Chairman, Code CS 1
   Naval Postgraduate School
   Monterey, CA 93943-5101

7. Prof. C. Thomas Wu ,CS/Wu 1
Naval Postgraduate School
Monterey, CA 93943-5100

8. LCDR Chris EAGLE ,CS/Ce 1
Naval Postgraduate School
Monterey, CA 93943-5100

9. Yazilim Gelistirme Grup Baskanligi 1
Deniz Harp Okulu Komutanligi
Tuzla
Istanbul, TURKEY

10. LTJG. Yildiray HAZIR 2
Yuksek Sokak – Onurkent Sitesi
D2 Blok D3
81570 - Kucukyali
Istanbul, TURKEY

63 290NPG TH 2442
6/02 22527-200 NLB

DUDLEY KNOX LIBRARY
3 2768 00403543 6